

AD-A040 468

GOODYEAR AEROSPACE CORP AKRON OHIO
SPECIFICATION FOR A STARAN PROGRAMMING LANGUAGE.(U)

F/G 9/2

UNCLASSIFIED

APR 77 R G LANGE, H CHEESEMAN, E W DAVIS

F30602-76-C-0200

GER-16347

RADC-TR-77-131

NL

1 OF 2
AD
A040468



AD A 040 468

RADC-TR-77-131
Final Technical Report
April 1977

12
nu



SPECIFICATION FOR A STARAN PROGRAMMING LANGUAGE

Goodyear Aerospace Corporation



Approved for public release; distribution unlimited.

AD No. _____
DDC FILE COPY

ROME AIR DEVELOPMENT CENTER
AIR FORCE SYSTEMS COMMAND
GRIFFISS AIR FORCE BASE, NEW YORK 13441

This report has been reviewed by the RADC Information Office (OI) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public including foreign nations.

This report has been reviewed and is approved for publication.

APPROVED:

Armand A. Vito

ARMAND A. VITO
Project Engineer

APPROVED:

Robert D. Krutz

ROBERT D. KRUTZ, Colonel, USAF
Chief, Information Sciences Division

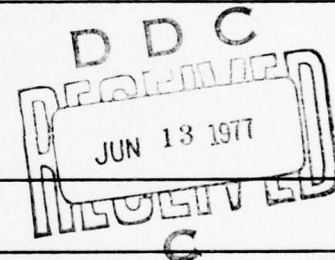
FOR THE COMMANDER:

John P. Huss

JOHN P. HUSS
Acting Chief, Plans Office

Do not return this copy. Retain or destroy.

19 REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADCTR-77-131	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) SPECIFICATION FOR A STARAN PROGRAMMING LANGUAGE	5. TYPE OF REPORT & PERIOD COVERED Final Technical Report, 8 Mar 76 - 8 Oct 76	
7. AUTHOR(s) R.G. Lange, E.W. Davis H. Cheeseman, R.W. Lott	6. PERFORMING ORG. REPORT NUMBER GER-16347	
PERFORMING ORGANIZATION NAME AND ADDRESS Goodyear Aerospace Corporation 1210 Massillon Road Akron OH 44315	8. CONTRACT OR GRANT NUMBER(s) F30602-76-C-0200	
9. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (ISCA) Griffiss AFB NY 13441	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS 62702F & 63701B 32020317	
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Same	12. REPORT DATE Apr 1977	
	13. NUMBER OF PAGES 171	
	15. SECURITY CLASS. (of this report) UNCLASSIFIED	
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report) Same		
18. SUPPLEMENTARY NOTES RADC Project Engineer: Armand A. Vito (ISCA) This document was prepared with the aid of the RUNOFF Facility at RADC.		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) Programming Language Higher Order Language (HOL) Parallel Processing High Level Language (HLL) Associative Processing Language Specification STARAN		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) This specification defines a procedure-oriented language with emphasis on parallel arithmetic and associative processing features. The meanings of the more common operations on scalars are promoted to multiple element data organizations. Data types and structures are provided to support these operations, and the structured design and programming disciplines. (see reverse)		



156800

Drac

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

The language is designed to be useful with the Goodyear Aerospace Corporation STARAN computer system. However, no feature of the language is strictly dependent on the STARAN architecture for proper execution.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

PREFACE

The facilities of the Multics system at Rome Air Development Center were used in the preparation of this report.

APPROVED BY		
DATE	APPROVED BY	<input checked="checked" type="checkbox"/>
DATE	APPROVED BY	<input type="checkbox"/>
DATE	APPROVED BY	<input type="checkbox"/>
DISTRIBUTION/AVAILABILITY CODES		
DATE	APPROVED BY	SPECIAL
A		

TABLE OF CONTENTS

1. INTRODUCTION	1-1
1.1 Purpose of the Specification	1-1
1.2 Organization of the Specification	1-1
1.3 Definitions	1-2
1.4 Syntax Meta-language	1-2
1.4.1 Category-name	1-3
1.4.2 Production Rules and Syntax	1-3
1.4.3 Syntactic Expressions and Units	1-4
1.4.4 Application of the Production Rules	1-4
1.5 Additional Conventions	1-6
1.6 Implementation Defined Properties	1-7
2. STATIC PROGRAM STRUCTURE	2-1
2.1 Programs	2-1
2.2 Blocks and Block Structure	2-3
2.3 Groups	2-3
2.4 Statements	2-4
2.4.1 Procedural Statements	2-4
2.4.2 Statement Prefixes	2-5
2.5 Lexical Syntax	2-5
2.5.1 Text	2-5
2.5.2 Delimiters, Blanks, and Comments	2-5
2.5.3 Identifiers	2-6
2.5.4 Literal Constants	2-7
2.5.5 I-subs	2-9
2.6 Reserved Words	2-9
2.7 Keyword Abbreviations	2-9
3. DATA	3-1
3.1 Variables and Constants	3-1
3.2 Data Types	3-1
3.2.1 Cardinal	3-2
3.2.2 Fixed-point	3-2
3.2.3 Floating-point	3-2
3.2.4 Bit-string	3-3
3.2.5 Character-string	3-3
3.3 Data Organization	3-3
3.3.1 Scalars	3-3
3.3.2 Arrays of Scalars	3-3
3.3.3 Structures	3-4
3.3.4 Arrays of Structures	3-4
3.3.5 Data Aggregates	3-4
3.3.6 Sharing of Storage	3-5
3.4 Data Storage	3-7

3.4.1	Storage Classes	3-7
3.4.2	Memory Properties	3-8
3.4.3	Storage of Arrays	3-8
3.4.4	Connected Storage	3-8
3.5	Declarations	3-8
3.5.1	Scope of a Declaration	3-8
3.5.2	The Declare Statement	3-9
3.6	The Data Attributes	3-10
3.6.1	Aligned	3-10
3.6.2	Automatic	3-11
3.6.3	Bit	3-11
3.6.4	Builtin	3-12
3.6.5	Cardinal	3-12
3.6.6	Character	3-12
3.6.7	Constant	3-12
3.6.8	Controlled	3-13
3.6.9	Defined	3-13
3.6.10	Dimension	3-13
3.6.11	Entry	3-14
3.6.12	External	3-14
3.6.13	Fixed	3-14
3.6.14	Float	3-15
3.6.15	Initial	3-15
3.6.16	Length	3-15
3.6.17	Memory	3-16
3.6.18	Precision	3-16
3.6.19	Range	3-16
3.6.20	Static	3-17
3.6.21	Unaligned	3-17
3.6.22	Varying	3-18
3.7	Data Attribute Sets	3-18
3.7.1	Conflicting Attribute Set Members	3-18
3.7.2	Required Attribute Set Members	3-19
3.7.3	Default Attributes	3-20
4.	DATA REFERENCES AND EXPRESSIONS	4-1
4.1	Expressions	4-1
4.1.1	Evaluation of Expressions	4-2
4.1.2	Evaluation of Prefix and Infix Expressions	4-2
4.1.3	The Order of Operator Evaluation	4-2
4.1.4	Reducibility and Side-effects	4-4
4.1.5	Exceptions in Expression Evaluation	4-4
4.2	References	4-5
4.2.1	Simple Reference	4-7
4.2.2	Array Reference	4-7
4.2.3	Selection Qualification	4-8
4.2.4	Structure Qualification	4-10
4.2.5	Qualified Reference Resolution	4-12
4.2.6	Function Reference	4-13

4.2.7	Built-in Function References	4-14
4.2.8	Parameters and Arguments	4-14
4.2.9	Function Reference Reducibility	4-16
4.3	Formal Syntax of Expressions	4-16
4.3.1	General Expressions	4-17
4.3.2	Constant-valued Expressions	4-17
4.4	Operators	4-18
4.4.1	Arithmetic Operators	4-18
4.4.2	Bit-string Operators	4-22
4.4.3	The String Operator	4-23
4.4.4	The Relational Operators	4-24
4.5	Conversions	4-25
4.5.1	Conversion Order	4-25
4.5.2	Cardinal to Fixed	4-26
4.5.3	Fixed to Cardinal	4-26
4.6	Conformance of Aggregate Types	4-27
4.6.1	Contexts That Force Promotion	4-27
4.6.2	Types of Promotion	4-28
4.6.3	Promotion Rules	4-28
4.7	Built-in Functions	4-29
4.7.1	String Built-in Functions	4-30
4.7.2	Arithmetic Built-in Functions	4-38
4.7.3	Mathematical Built-in Functions	4-47
4.7.4	Array Built-in Functions	4-49
4.7.5	Miscellaneous Built-in Functions	4-49
5.	THE DYNAMIC BEHAVIOR OF A PROGRAM	5-1
5.1	The Flow of Control	5-1
5.2	A Program	5-1
5.3	Procedures	5-1
5.3.1	Procedure Statement	5-3
5.3.2	The Routine Statement	5-4
5.3.3	Entry Attributes	5-5
5.4	Procedures and Dynamic Block Structures	5-7
5.4.1	Procedure Activation	5-7
5.4.2	The Environment of an Activation	5-7
5.5	The Flow of Control Within an Activation	5-8
5.6	Inter-block Flow of Control	5-8
5.6.1	Procedures	5-8
5.6.2	On Units	5-9
6.	CONTROL STATEMENTS	6-1
6.1	Iterative DO-Loop	6-1
6.1.1	Range of a DO-Loop	6-1
6.1.2	Active and Inactive DO-Loops	6-2
6.1.3	Executing an Iterative DO Statement	6-2
6.1.4	Loop Control Processing	6-2
6.1.5	Execution of the Range	6-3
6.1.6	ENDDO Statement Processing	6-3

6.1.7	Incrementation Processing	6-3
6.2	Repetitive DO-Loop	6-4
6.2.1	Range of a DO WHILE-loop	6-4
6.2.2	Active and Inactive DO WHILE-loops	6-5
6.2.3	Executing a DO WHILE Statement	6-5
6.2.4	Loop Control Processing	6-5
6.2.5	Execution of the Range	6-5
6.2.6	ENDDO Statement Processing	6-6
6.3	The Do Case Group	6-6
6.3.1	Body of DO CASE-group	6-7
6.3.2	Executing a DO CASE Statement	6-7
6.3.3	Transfer Relative to a DO CASE-group	6-7
6.4	Conditional Statements	6-8
6.4.1	The IF Statement	6-8
6.4.2	The IFARRAY Statement	6-9
6.4.3	The Logical If Statement	6-11
6.5	CALL and RETURN Statements	6-11
6.5.1	CALL Statement	6-11
6.5.2	RETURN Statement	6-12
6.6	GO TO Statement	6-12
6.7	EXIT Statement	6-13
7.	ASSIGNMENT AND ALLOCATION STATEMENTS	7-1
7.1	Assignment Statement	7-1
7.1.1	Scalar Assignment	7-1
7.1.2	Array Assignment	7-3
7.1.3	Scalar Broadcast Assignment	7-3
7.1.4	Structure Assignment	7-4
7.2	ALLOCATE Statement	7-4
7.3	FREE Statement	7-5
8.	INPUT/OUTPUT	8-1
8.1	Introduction	8-1
8.2	Datasets	8-1
8.2.1	Dataset Organization	8-2
8.2.2	Consecutive Dataset	8-2
8.2.3	Direct dataset	8-2
8.3	Files	8-2
8.3.1	File Attributes	8-2
8.3.2	Opening a file	8-5
8.3.3	The Open Statement	8-6
8.3.4	Closing a File	8-7
8.3.5	The Close Statement	8-8
8.4	Data Transmission statements	8-8
8.4.1	The Read Statement	8-8
8.4.2	The Write Statement	8-10
8.4.3	The Rewrite Statement	8-11
8.4.4	The Delete Statement	8-12
8.4.5	Transmission Options	8-13

8.4.6 Data Transmission Examples	8-15
9. CONDITIONS	9-1
9.1 Conditions in the Language	9-1
9.1.1 Conversion Condition	9-1
9.1.2 Endfile Condition	9-1
9.1.3 Error Condition	9-2
9.1.4 Finish Condition	9-2
9.1.5 Fixedoverflow Condition	9-2
9.1.6 Key Condition	9-2
9.1.7 Overflow Condition	9-2
9.1.8 Record Condition	9-2
9.1.9 Size Condition	9-3
9.1.10 Storage Condition	9-3
9.1.11 Stringrange Condition	9-3
9.1.12 Stringsize Condition	9-3
9.1.13 Subscriptrange Condition	9-3
9.1.14 Transmit Condition	9-3
9.1.15 Undefinedfile Condition	9-3
9.1.16 Underflow Condition	9-4
9.1.17 Zerodivide Condition	9-4
9.2 Statements and Program Effects	9-4
9.2.1 The Signal Statement	9-4
9.2.2 The Revert Statement	9-5

Appendices

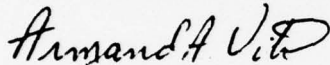
Complete Syntax	A-1
Terms	B-1
Sample Programs	C-1
User Library Sample Procedure	D-1

EVALUATION

This effort was undertaken to provide a specification of a Higher Order Language (HOL) for the Goodyear Aerospace Corporation STARAN parallel, associative, array computer. This would allow programmers to apply parallel, associative and array solutions to computational problems using a HOL.

The benefits of a HOL for a STARAN type architecture are the same as for a serial computer with the additional benefit that it will aid the programmer in his thinking toward parallel and associative solutions for his computational problems. The effort represents the successful completion of one aspect of RADC's comprehensive investigation into associative processing for Air Force applications (TPO-12, FY 75 & 76).

This was a joint effort by RADC and the Defense Mapping Agency (DMA). RADC and DMA have a similar desire to derive computational benefits through the use of parallel, associative, array architectures.



ARMAND A. VITO
Project Engineer

1. INTRODUCTION

This specification document defines a language with parallel arithmetic and associative processing features. Statements are defined to support the design of programs with arithmetic, logical, and associative processing aspects. The meanings of operations and expressions extend over single and multiple element data organizations. The types, organization, and attributes of data are versatile in their support of the language features.

The language is designed to be useful with the Goodyear Aerospace Corporation STARAN (1) computer system. However, no feature of the language is strictly dependent on the STARAN architecture for proper execution.

1.1 Purpose of the Specification

This specification contains the syntactic and semantic definitions of the language. A formal meta-language is used to define the syntax. Semantic definitions are in prose. Sufficient detail is provided for developing a compiler for the language.

Information necessary to write programs is included, however this specification is not intended to be a language reference manual. It does not include programming examples unless they are useful for specification purposes. It does not attempt to show total application program development.

1.2 Organization of the Specification

Section 1, Introduction, gives information which is useful or necessary for understanding the remainder of the specification. The formal notation used throughout the document is defined. No actual language details are introduced. Section 2, Static Program Structure, gives both the highest and lowest level language definition. At the highest level, program structure is established. At the lowest level, the lexical syntax is defined.

Section 3 is concerned with data. The data types, organization, and attributes are specified. Section 4, Data References and Expressions, covers the uses of data as operands in expressions and the operators in the language are defined. Dynamic Program Execution, section 5, is concerned with the run time definitions of procedure invocation, parameter passing, block activation, and flow of control.

(1) TM Goodyear Aerospace Corp., Akron, Ohio 44315

Specification of control statements is given in section 6. Assignment and storage allocation statements are covered in section 7. In section 8 the definition of files and the input/output statements are given. Section 9 defines the facilities for signalling a process based on the existence of a particular condition or process interrupt.

1.3 Definitions

Terms defined in prose are underlined when defined and not underlined thereafter. Examples are provided to aid understanding but are not intended to be comprehensive or definitive.

All examples are clearly set off from the rest of the text as shown by the example below. Within examples where empty space might be misleading, \emptyset denotes a blank. Upper and lower case letters are used freely in examples to help distinguish declared variables from keywords of the language.

1.4 Syntax Meta-language

The language is defined using a formal meta-language to define the syntax and prose to describe the semantics. Although this is not a strictly formal definition, both the syntactic and semantic descriptions are reasonably precise and complete.

The syntax of the language is defined by a set of syntax rules expressed in a formal notation derived from Backus-Naur Form (BNF). Each syntax rule describes a character-string or pattern of characters that constitutes a syntactic construct of the language proper. The complete set of syntax rules describes all syntactically correct programs in the language.

Each rule is of the form:

category-name ::= one or more syntactic
expressions

Example:

<file option> ::= FILE (<expression>)

In the example, <file option> is a category-name that represents the character-string described by the syntactic expression on the right of the definition symbol "::<=". "FILE" is a notation constant that represents an actual occurrence of the character-string "FILE". The occurrence of <expression> is a category-name defined by another rule. The occurrence of (

and) represent actual occurrences of left and right parenthesis.

1.4.1 Category-name

A category-name is bracketed by < and > and contains any sequence of lower case letters and other characters not including >.

Example:

```
<into option> ::= INTO ( <reference> )
```

When the prose refers to the category-name <into option> or <reference>, these terms appear exactly as they do in the syntax rule. When a keyword, such as INTO, appears in the prose, it is enclosed in quotes("") or written as upper case to delimit it from the prose.

1.4.2 Production Rules and Syntax

Production rules serve as the declarative portion of the metalanguage and do so by specifying restrictions on the forms that are acceptable program text.

A production-rule consists of a category-name, followed by the symbol "::=" and then followed by a syntactic-expression (see Section 1.4.3). The symbol "::=" can be read as "is defined to be".

Such a production-rule is termed a defining production-rule for the category-name written before the "::=". Within this document, there is at most one defining production-rule for any given category-name.

The basic function of a production-rule is to define a set of possibilities for the number, type(s), and order of the immediate subnodes of a node in the syntax tree whose type is the defined category-name. This is done by interpreting the syntactic-expression of the production-rule according to the algorithms given below (see Section 1.4.4).

A syntax is any set of production-rules. For example, the set of all of the production-rules in this document is a syntax.

If a defining production-rule for a category-name occurs in a syntax, then that category-name is said to be non-terminal. Any category-name that occurs somewhere within the syntactic-expressions of the production-rules of the syntax, but has no defining production rule in the syntax, is said to be terminal.

1.4.3 Syntactic Expressions and Units

Given a syntax, a syntactic-expression is defined to be either a single syntactic-unit, or several syntactic-units any of the adjacent pairs of which is possibly separated by a "|" or a "■". The symbols are called the or-symbol and the bullet, respectively.

Given a syntax, a syntactic-unit is defined to be one of the following:

a single category-name,

a syntactic-expression enclosed in the brackets "[" and "]",

a syntactic-expression enclosed in the braces "{" and "}", or

a syntactic-expression followed by an ellipsis "...".

1.4.4 Application of the Production Rules

Given a syntax and a category-name, the algorithm shown just below obtains an (possibly empty) ordered set of category-names, termed here an interpretation with respect to the given syntax of the given category-name.

1.4.4.1 Category-name Meaning

An interpretation of a category-name is defined as follows:

Case 1.

There is in the given syntax a defining production rule for the given category-name.

An interpretation is an interpretation of the syntactic-expression written following the "::<=" in the defining production-rule.

Case 2.

(Otherwise).

The given category-name is a terminal with respect to the given syntax; the interpretation is the empty set.

1.4.4.2 Syntactic Expression Meaning

An interpretation of a syntactic-expression is defined as follows:

Case 1. The syntactic-expression is a syntactic-unit.

Case 1.1 One Instance

The syntactic-unit is a single category-name.

An interpretation consists of the ordered set containing just this single category-name.

Case 1.2 Option

The syntactic-unit is a syntactic-expression enclosed in the brackets "[" and "]".

An interpretation consists either of an interpretation of the enclosed syntactic-expression, or of the empty set.

Case 1.3 Grouping

The syntactic-unit is a syntactic-expression enclosed in the braces "{" and "}",

An interpretation consists of an interpretation of the enclosed syntactic-expression.

Case 1.4 Repetition

The syntactic-unit is a syntactic-expression followed by an ellipsis "...".

An interpretation consists of one or more interpretations of the given syntactic-expression.

Case 2.

The syntactic-expression is a sequence of two or more syntactic-units possibly separated by a "|" or a "■".

Case 2.1 An or-symbol occurs as at least one such a separator.

Consider all or-symbols occurring in the given syntactic-expression to partition it into a sequence of inner syntactic-expressions. An interpretation is one of any of these inner syntactic-expressions chosen arbitrarily.

Case 2.2 A bullet occurs as such a separator and an or-symbol does not.

Consider all bullets occurring thus in the given syntactic-expression to partition it into a sequence of inner syntactic-expressions. An interpretation is the same as one of a syntactic-expression formed by arranging the sequence in an arbitrary order and omitting these bullets.

Case 2.3 (Otherwise).

(The syntactic-expression is a sequence of syntactic-expressions optionally separated by blanks.)

An interpretation consists of the concatenation, in order, of interpretations of the syntactic-expressions of the sequence.

In summary the syntactic expressions are built using the following meta-operations.

Syntactic Expression Operation	Symbol
Catenation	"spaces"
Alternation	
Repetition	...
Grouping	{ }
Optional	[]
Unordered Catenation	■

Note:

In the language specification the use of left-recursive production rules for category-names is avoided, although that form with the additional category-names as required is useful for implementation.

1.5 Additional Conventions

Common mathematical symbols are used with their usual meaning. In addition, the following notational conventions are used:

* denotes multiplication;
 / denotes division;
 ceil(x) denotes the smallest integer
 larger than or equal to x;
 floor(x) denotes the largest integer
 smaller than or equal to x;
 min(x,y) denotes the value of x if $x \leq y$,
 otherwise the value of y;
 max(x,y) denotes the value of x if $x \geq y$,
 otherwise the value of y.

1.6 Implementation Defined Properties

The following list shows those items or situations in the language which are defined by the implementation rather than by the language. They may be user controllable or absolutely bound by the implementation.

1. Determination of actual dataset names (8.3.2) on file opening.
2. ENVIRONMENT attribute and option syntax and semantics (8.3.1.9).
3. Standard action for conditions (9.2).
4. Situations when STORAGE condition is raised (9.2).
5. Actual representation of a dataset.
6. The size of a RECORD.
7. Situations in which TRANSMIT condition is raised.
8. Raising of RECORD condition by WRITE statement.
9. Position of records in a KEYED SEQUENTIAL file.
10. Length of a record KEY (8.3.1.8).
11. Maximum precisions of arithmetic data (4.4.1).
12. Determination of floating-point results of expressions and builtin functions (4.4.1.2, 4.7).
13. Character collating sequence (4.7, COLLATE).
14. Exact results of numeric conversions (4.7).
15. Number of digits (bits) in the exponent of a floating-point data item.
16. Situations when ERROR is raised.

The format of DEFINED is:

DEFINED <reference>

The <extent> expressions of a defined variable are evaluated upon block activation and saved in the block activation record; see section 3.6.2 for the rules on the extents of automatic variables.

A defined variable is always associated with the generation identified by its <reference> -this is known as its base <reference>. The defined variable is never allocated and thus has no <initial> attribute. The variable identified by the base <reference> cannot be a defined variable or named constant (<constant> attribute).

The <defined attribute> cannot be specified for members of structures. When specified for a structure, it maps the entire structure onto the generation of storage identified by the base <reference>.

The <expression>s contained in the base <reference> are evaluated upon each reference to the defined variable. Any <reference>s within the base <reference> are resolved in the <block> in which the defined variable is declared. Refer to section 4 for a complete discussion of <reference> resolution and evaluation.

Both the extents of the defined variable and those of the base variable, are used to determine if any condition has occurred; these are subscriptrange, stringrange, and stringsize (refer to section 9).

3.3.6.3 I-sub Defining

I-sub defining allows an array to be defined onto another array by means of a programmer-defined mapping between the elements of the defined array and its base array.

For i-sub defining the <reference> must contain one or more <i-sub>s in its <subscript>s.

For i-sub defining, the form of the <reference> defines the relationship between elements of the defined variable and elements of the base variable. The subscript expression selects an element of the base variable for each allowable combination of subscripts of the defined variable. If SUBSCRIPTRANGE is enabled for a reference to the defined variable, the subscripts are checked against the bounds of the defined variable, then the i-sub expression is evaluated and the resulting subscripts are checked against the bounds of the base variable. Each <subscript> value is converted to a binary integer (FIXED(15,0)) before replacing an <i-sub>.

The aggregate type of an expression is the dimensionality, array extents, and structuring determined by the rules of expression evaluation and aggregate promotion given in section 4.

3.3.6 Sharing of Storage

There are two facilities specified in the language for sharing a generation of storage among two or more variables. These are parameters and defined variables. These features require that the variables which share a generation of storage have identical data types and alignment <attribute>s, thus providing a guarantee that the variables have identical storage representations.

Both facilities provide techniques for sharing a generation of storage that is contained in an aggregate generation without having to share the entire aggregate. For example, scalars can be mapped onto array elements or members of structures and arrays can be mapped onto portions of arrays or arrays of structures.

3.3.6.1 Storage Sharing by Parameters

There are several possible forms of parameter passing in programming languages. This specification of argument passing in section 4 describes argument passing by-value and by-reference. When a variable is passed by-reference to a parameter, the variable and the parameter refer to the same generation of storage and thus share that generation.

Example:

```
sub: PROCEDURE(p); ... ENDPROC sub;  
    CALL sub(a);
```

The execution of the CALL statement invokes sub. During the block activation of SUB A and P both refer to the same generation of storage.

3.3.6.2 Array Storage Sharing by Defined Variables

The <defined attribute> is provided with "i-sub defining" to map a defined variable onto a generation of storage of another variable. The latter may be an array or an array of structures.

3.3.3 Structures

A structure is an ordered set of data elements which may have different attributes but must have <level> specified in the <declaration component>s. The elements may be scalars or arrays. A hierarchical relationship exists among data elements of the structure. The relationship is indicated by the relative values of the level numbers in the declaration.

For a given pair of <declaration component>s with equal <level> numbers, the data items are on the same hierarchical level. If the second <level> number is larger, its <declaration component> is nested within the first. If the second <level> number is smaller than the first, the first <declaration component> is nested within the second. The main structure (root) is a <declaration component> with <level> that is not subordinate to any <declaration component> and is said to be (hierarchically) at level-one. Nested structures are the subordinate <declaration component> items and are at higher numbered "level"s.

A variable is defined to be a level-one variable if it is a main structure or a declared variable that is not a structure. Thus scalars and arrays which are not in a structure are level-one variables.

The main structure, nested structures, and the data elements have names. Qualification is used to uniquely identify a nested structure or data element name. A qualified name consists of the name of the main structure and all nested structures leading to the structure or data element name to be identified, with periods between each pair of names.

3.3.4 Arrays of Structures

An array of structures is a structure with the attribute <dimension>. Each element of the array is thus a structure with structuring identical to all other elements.

3.3.5 Data Aggregates

An aggregate value is a set of scalar values. An aggregate value is an array of scalars, a structure, or an array of structures. Named constants, variables, functions, and expressions can have aggregate values.

The data type of an aggregate value is the ordered set of data types of its scalar components. The aggregate type of an aggregate named constant, variable, or function value is the dimensionality and array extents specified by the <dimension> attribute and the structuring specified by the <level>s used in its declaration.

3.2.4 Bit-string

A bit-string data item represents a bit-string value stored as a contiguous sequence of bits.

The attribute for declaring a bit-string data item is <bit>. The <length> attribute is used to specify the number of bits in the sequence.

3.2.5 Character-string

A character-string data item represents a character-string value stored as a contiguous sequence of characters. Character data is represented in the ASCII code.

The attribute for declaring a character-string item is <character>. The <length> attribute is used to specify the number of characters in the string.

3.3 Data Organization

The language has several modes for data items, either single items called scalars or collections. The latter are referred to collectively as aggregates.

3.3.1 Scalars

A scalar is a single element of data or one member of a set of data elements. A scalar may appear in a program as a constant or as a variable representing one element of data.

3.3.2 Arrays of Scalars

An array is an ordered set of scalars, all having identical attributes. An array is identified by a single symbolic name. An array appears in a program as a variable representing a set of scalars.

An array element is a scalar. The unique identification of an array element consists of the array name and the position of the element in the array. The position in the array is indicated by a bracketed subscript list following the name as specified in section 4.2.

The elements of an array are stored as an ordered sequence such that the rightmost subscript varies most rapidly and the leftmost subscript varies least rapidly. This order is called row-major order and is consistent with arrays of structures, section 3.3.4.

3.2.1 Cardinal

A cardinal data item represents an unsigned integral value stored as a binary number; therefore, cardinal data may assume only zero or positive integral values.

The attribute for declaring a cardinal data item is `<cardinal>`. The `<precision>` attribute is used to specify the number of bits of precision for the data item.

3.2.2 Fixed-point

A fixed-point data item represents a signed, real value stored as a binary number with an assumed binary point. Fixed-point data may take on negative, zero, or positive values.

The attribute for declaring a fixed-point data item is `<fixed>`. It must be followed by the `<precision>` attribute or the `<range>` attribute to define the set of values for the data item.

The `<precision>` attribute is used to specify the number of bits of precision for the data item and the scale factor. The scale factor indicates the location of the assumed binary point.

The `<range>` attribute, if written, specifies the smallest and largest values for the fixed-point data and the significance required for intermediate values. The latter is determined by the `<scale factor>` as a function of the `<scale>` option of the `<range>` attribute (see section 3.6.19).

A fixed-point integer is a fixed-point data item with a scale factor of zero.

3.2.3 Floating-point

A floating-point data item represents a signed, real value stored as a signed mantissa with a signed exponent.

The attribute for declaring a floating-point data item is `<float>`. The `<precision>` attribute is used to specify the number of bits of precision for the mantissa of the data item.

3. DATA

3.1 Variables and Constants

A variable is a data item which may change in value during the execution of a program. A constant is a data item which cannot change in value. Reference to a data item is made by the use of a symbolic name or the occurrence of a literal constant in a program. A variable appears in a program as a symbolic name. A constant appears in a program either as a literal constant or as a named constant. The syntax for literal constants is specified in section 2. Named constants are defined with the <constant> attribute.

3.2 Data Types

Data is separated into computational and non-computational categories. The types of data in the non-computational category are "entry" and "file". They are primarily specified in sections 5 and 8, respectively. Five types of computational data are defined. Each type is different in internal representation and in the values it may assume. The specification of data types is concerned with the abstract properties of the data rather than the internal representation. Thus the storage requirements for each type are not specified.

Computational data is further separated into arithmetic and string types.

Arithmetic data types are:

- (1) cardinal
- (2) fixed-point
- (3) floating-point

String data types are:

- (1) bit-string
- (2) character-string

Arithmetic data is represented in a binary base. String data is a contiguous sequence of bits or characters.

The term extent is used to reference the precision of arithmetic data, the length of string data, and/or the dimensions of an array.

Static Program Structure

ALLOCATION	ALLOCN
CHARACTER	CHAR
CONDITION	COND
CONTROLLED	CTL
ENVIRONMENT	ENV
FIXEDOVERFLOW	FOFL
NOCONVERSION	NOCONV
NOFIXEDOVERFLOW	NOFOFL
NOOVERFLOW	NOOFL
NOSTRINGRANGE	NOSTRG
NOSTRINGSIZE	NOSTRZ
NOSUBSCRIPTRANGE	NOSUBRG
NOUNDERFLOW	NOUFL
NOZERODIVIDE	NOZDIV
PRECISION	PREC
SEQUENTIAL	SEQL
STRINGRANGE	STRG
STRINGSIZE	STRZ
SUBSCRIPTRANGE	SUBRG
UNALIGNED	UNAL
UNDERFLOW	UFL
UNDEFINEDFILE	UNDF
ZERODIVIDE	ZDIV

The <exponent> of a decimal <floating-point constant> denotes a power of ten.

2.5.5 I-subs

An <i-sub> is a <token> used only in a <subscript> of a <base reference> in a <defined> attribute. Its semantics are described in section 3.3.6.

Syntax:

<i-sub> ::= <decimal integer> SUB

Example:

1SUB

2.6 Reserved Words

A small number of the language keywords are reserved and may not be declared as user names. The reserved keywords are primarily those related to program structure.

PROCEDURE	ENDPROC	DO	ENDDO	
ON	ENDON	ROUTINE		
IF	IFARRAY	THEN	ELSE	ENDIF
DECLARE	SUB	CASE	ENDCASE	

2.7 Keyword Abbreviations

Abbreviations are defined for certain keywords and builtin-functions. The abbreviations themselves are keywords or builtin-functions and will be recognized as synonymous in every respect with the full denotations, except that in the case of builtin-functions the abbreviations have separate declarations and name scopes. The abbreviations are shown to the right of the full denotations in the following list.

A <cvi expression> is defined to be a constant-valued expression (see section 4.3.2).

Syntax:

```
<character-string constant> ::= [ (<cvi expression>) ]
                                ' [<character>]... '
```

A null character string is denoted by "". The character ' is represented within a character string by "".

Example:

Source string	Character value
'abcd'	abcd
''''	
'abcd''ef'	abcd'ef
'''ab'''	'ab'

2.5.4.3 Arithmetic Constants

An <arithmetic constant> is defined to be an arithmetic value that remains constant and cannot be changed. The attributes of the constant, i.e., type and precision, are normally determined by the constant's syntax.

Syntax:

```
<arithmetic constant> ::= <decimal constant>
                           | <binary fixed-point constant>
<binary fixed-point constant> ::=
    {0|1}... [ . [ {0|1}... ] ] B
    | . {0|1}... B
<decimal constant> ::= <decimal fixed-point constant>
                       | <floating-point constant>
<floating-point constant> ::= <mantissa><exponent>
<mantissa> ::= <decimal fixed-point constant>
<exponent> ::= E[+ | -]<decimal integer>

<decimal fixed-point constant> ::=
    <decimal integer>
    [ . [<decimal integer>] ]
    | . <decimal integer>
<decimal integer> ::= <digit>...
```

2.5.4 Literal Constants

A <literal constant> is used for an arithmetic or string value.

Syntax:

```
<literal constant> ::= <bit-string constant>
                        | <character-string constant>
                        | <arithmetic constant>
```

2.5.4.1 Bit-String Constants

A <bit-string constant> is defined to be a sequence of zero and one bits derived from the zero and one binary digits of the constant within quotes. If the optional <cvi expression> is present, the string of zero and one bits is concatenated to itself N-1 times, where N is the value of the <cvi expression> and is an integer greater than zero.

Syntax:

```
<bit-string constant> ::= [ (<cvi expression>) ]
                        {  B ' <bit chars>... '
                        |  Q ' <quad chars>... '
                        |  O ' <octal chars>... '
                        |  X ' <hex chars>... ' }

<bit chars> ::=      0 | 1 | <space>
<quad chars> ::=     2 | 3 | <bit chars>
<octal chars> ::=    4 | 5 | 6 | 7 | <quad chars>
<hex chars>  ::=     8 | 9 | A | B | C | D | E | F
                        | <octal chars>
```

The four forms of <bit-string constant> allow 1, 2, 3, or 4 bits to be specified by each source "digit". The <space>s can be used to format longer constants for readability.

2.5.4.2 Character-String Constants

A <character-string constant> is defined to be the sequence of characters derived by substituting a single quote for all double quotes, and deleting the containing quotes. If the optional <cvi expression> is present, the resulting string of characters is concatenated to itself N-1 times, where N is the value of the <cvi expression> and is an integer greater than zero.

Syntax:

```

<delimiter> ::= <graphic delimiter> | <space> | <comment>
<graphic delimiter> ::= + | - | * | / | ~ | &
                        | : | ; | ( | ) | _ | ||
                        | , | . | | | |
                        | = | ~= | ≤ | ≥ |
                        | ≤ | ≥ | ~≤ | ~≥

<space> ::= <blank> | <newline> | <newpage> | <tab>
<blank> ::= ASCII blank character
<newline> ::= ASCII new line character
<newpage> ::= ASCII new page character
<tab> ::= ASCII horizontal tab or ASCII vertical tab

<comment> ::= /* ASCII characters except an asterisk
              followed by a slash */

```

The <space>s and <comment>s can be used anywhere without restriction on the length of a <comment> or the number of <space>s between any two <token>s. Whenever adjacent <identifier>s or <literal constant>s occur in the syntax rules, at least one <space> or <comment> must be used to separate them.

The individual usage and semantics for the <graphic delimiter>s are given in subsequent sections of the specification.

2.5.3 Identifiers

Identifiers are used for <declared name>s or keywords. A keyword introduces a statement, a statement option, or a builtin function.

Syntax:

```

<identifier> ::= <letter>
                [ <letter> | <digit> | _ | $ ]...

<letter> ::= A | B | C | D | E | F | G | H | I
            | J | K | L | M | N | O | P | Q | R
            | S | T | U | V | W | X | Y | Z

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

```

A single <identifier> cannot be more than 32 characters long.

2.4.2 Statement Prefixes

Syntax:

```
<prefix> ::= <label prefix> | <case prefix>
<label prefix> ::= <identifier>:
<case prefix> ::= CASE ( <case number> ) :
<case number> ::= <cvi expression>
                | <cvi expression> : <cvi expression>
```

A <label prefix> is a means of naming a statement. In certain contexts, such as within a loop, any <basic statement> may be named by being preceded by a <label prefix>. A <cvi expression> is defined to be a constant-valued expression (see section 4.3.2).

2.5 Lexical Syntax

2.5.1 Text

The lowest level syntactical component of the language is called a <token>. Sequences of <token>s form statements which in turn form <program unit>s.

Syntax:

```
<token> ::= <identifier> | <literal constant>
          | <delimiter> | <i-sub>
```

2.5.2 Delimiters, Blanks, and Comments

2.4 Statements

The full syntax and semantics of the statements of the language are given in sections 3 and 5 through 9. In this section, only the general nature of statements is discussed and references to other relevant sections of this document are provided.

There are two types of statements: declarative statements and procedural statements. There is one declarative statement, the <declare statement>, fully discussed in section 3. The execution of a <declare statement> or <null statement> has no effect, even though all statements are considered to be executable.

2.4.1 Procedural Statements

The procedural statements are used to form the executable statements of a <program unit> in accordance with the syntax rules in section 2.2. Some of them represent individual statements, whereas the <if statement>, for example, must include a corresponding <endif>. The individual statements are called <single statement>s and are listed below.

Syntax:

```

<single statement> ::= <allocate statement>
                      | <assignment statement>
                      | <call statement>
                      | <close statement>
                      | <delete statement>
                      | <free statement>
                      | <goto statement>
                      | <null statement>
                      | <open statement>
                      | <read statement>
                      | <return statement>
                      | <revert statement>
                      | <rewrite statement>
                      | <signal statement>
                      | <write statement>

```

The ";" symbol is used to delimit statements.

Example:

```

MAIN: PROCEDURE;
:
:
SECOND: PROCEDURE;
:
:
ALT: ROUTINE;
:
:
UTILITY: PROCEDURE;
:
:
ENDPROC UTILITY;
:
:
ENDPROC SECOND;
:
:
ENDPROC MAIN;

```

MAIN is the <external procedure> and contains one <procedure> named SECOND at lexical level two. The latter has another routine (an entry) ALT, and also contains a <procedure> named UTILITY at block level three.

2.3 Groups

A <group> is a list of statements that are bracketed to control the program flow during execution.

Syntax:

```

<group> ::= <iterative loop>
          | <repetitive loop>
          | <case group>

```

The semantics of <group>s are contained in section 6.

Each <group> is headed by a form of the DO statement. The DO statement is the keyword DO followed by an iterative loop specification, a repetitive loop specification or a case selection expression.

Syntax:

```

<block> ::= <procedure>
<procedure> ::= <procedure statement>
                <procedure body>
                <endproc statement>
<procedure body> ::= <procedure component>
                    [ <routine statement>
                      <procedure component> ]...
<procedure component> ::= <statement>...
<statement list> ::= <statement>...
<statement> ::= <procedure>
                | <declare statement>
                | <basic statement>
                | <prefix><basic statement>
<basic statement> ::= <group>
                    | <independent statement>
<independent statement> ::= <single statement>
                            | <conditional statement>
                            | <on statement>

```

All of the text of a <procedure>, except the <label prefix>s of each of its <routine statement>s and the <closure identifier> of its <endproc statement>, is contained in the <procedure>.

Example:

```

LOOK:  PROCEDURE;
      :
      :
      _____
SEARCH: ROUTINE;
      :
      :
      _____
      ENDPROC LOOK;

```

The text shown with lines is contained in <procedure> LOOK.

The text contained in <block> LOOK, but not contained in any other <block> contained in LOOK, is immediately contained in <block> LOOK.

2. STATIC PROGRAM STRUCTURE

2.1 Programs

A `<program>` consists of one or more `<program unit>`s together with their environment; the latter is an implementation dependent context which includes a file system. A `<program unit>` is the largest syntactic construct of the language; it is also the unit of input for compilation.

The set of `<program unit>`s that constitute a `<program>` is determined dynamically by CALL statements and function references during execution of the program as described in section 5, or by the use of a linking facility prior to or during execution. The use of the latter would restrict the set of `<program unit>`s to a statically specified set.

Syntax:

```
<program> ::= <program unit>...  
<program unit> ::= <external procedure>  
<external procedure> ::= <procedure>
```

An `<external procedure>` is one not contained in another procedure. A `<program unit>` is always an external `<block>`.

2.2 Blocks and Block Structure

A `<block>` delimits the scope of names in an `<external procedure>` and is the major unit that determines the flow of control during program execution. Refer to section 3 for the description of the scope of names and to section 5 for the specification of the flow of control.

A `<block>` is an entire `<program unit>` (`<external procedure>`) or any `<procedure>` contained in another `<procedure>`.

- The order in which operands are promoted to higher aggregate types.
- The order in which the scalar data elements of aggregate operands are operated upon by infix or prefix operators.
- The order in which <function reference>s are evaluated and number of times a reducible <function reference> is evaluated.

4.1.4 Reducibility and Side-effects

A <procedure> may contain expressions which are identical in effect. If, during execution of the program, the values of the operands are not modified between evaluation of the expressions, such expressions are common expressions.

If the value of an operator can be determined without evaluation of one or more of its operands and no operand contains irreducible <function reference>s, the operands are not necessarily evaluated.

A <procedure> that depends on the full evaluation of all operands or upon an operand not being evaluated, is in error and the result of its execution is undefined.

Example:

These statements are in error:

if $B=0 \mid 1/E > N$ then ...

if $\text{length}(S)=0 \mid \text{substr}(S, \dots)$ then ...

4.1.5 Exceptions in Expression Evaluation

The facility of <on unit>s and the <on statement> allow programmer specification of certain exceptions that can occur during the evaluation of references and expressions. Refer to section 9.1 for a full discussion of conditions.

Since the order of <expression> evaluation is not strictly defined, the order in which conditions are detected and the frequency with which they occur are not defined.

If a condition is signaled during <expression> evaluation, the latter is suspended as it would be for the execution of a <signal statement>. See the description in section 9 regarding whether control can return from the <on unit> to the point the exception occurred. For most conditions control

Associative operators of equal precedence are evaluated in an unspecified order to allow optimization. The operators with this property are: + * | & . Nonassociative operators of equal precedence are evaluated from left-to-right, except for prefix operators which are evaluated from right-to-left.

For an <expression> the precedence of all operators is:

High		~	prefix +	prefix -	**
		*	/		
		infix +	infix -		
		=	~=	<	~< > ~> <= >=
		&			
Low					

Implicit conversions are not provided generally, only a subset of the above operators are allowed in a given context according to the data types of the operands (see section 4.5.1). For example, if all <reference>s have the data type bit-string, the operators allowed and their precedence is:

High		~					
		=	~=	<	~< > ~>	<=	>=
		&					
Low							

If all the data types are arithmetic, the operators allowed and their precedence is:

High		prefix +	prefix -	**
		*	/	
		infix +	infix -	
Low		=	~=	< ~< > ~> <= >=

The order of operator evaluation within <expression>s is determined by the precedence of operators (section 4.3) and by parenthesization. Otherwise the order of evaluation is not defined. A procedure that depends on any of the following properties is in error and the results of its execution are not defined:

- The order in which associative operators of equal precedence are evaluated.

4.1.1 Evaluation of Expressions

Basic expressions contain only a <reference> or a <literal constant>. The evaluation of a basic expression produces a value with aggregate type and data type determined by the declaration of the name identified by the <reference> or the declaration of the <literal constant>. If the basic expression is a <function reference>, the aggregate type and data type are the aggregate type and data type of the value returned by the function, that is, those declared with the function entry.

The basic expression evaluation produces the value of the variable or constant identified by the <reference> or <literal constant>. If the basic expression is a <function reference>, the value of the evaluated basic expression is the value returned by the function.

4.1.2 Evaluation of Prefix and Infix Expressions

The operator evaluation consists of:

1. The evaluation of the operand(s).
2. For infix expressions the two operands are promoted to the higher common aggregate type as described in promotion, section 4.6.
3. The conversion of the value of the evaluated operand(s) to the type required by the operator (if automatic conversion is defined, see section 4.5). If either promoted operand is an aggregate, the operator is applied to each (corresponding) scalar component in an unspecified order.
4. The application of the operator to the converted value of the operand. If the operand is an aggregate, the operator is applied to each scalar component of the aggregate.

The result of the evaluation of a prefix expression is a value whose aggregate type is the aggregate type of the evaluated operand. The data type of each scalar component of the result is the data type of the corresponding scalar component of the operand.

4.1.3 The Order of Operator Evaluation

Operator evaluation in an <expression> is performed in an order determined by the syntactic precedence of the operators (see 4.3). It can be overridden as needed by use of parentheses around subexpressions within the expression.

4. DATA REFERENCES AND EXPRESSIONS

An expression represents an algorithm used for computing a value. Expressions are of three types: scalar, array, and structure, depending upon the aggregate type of the result. An array (or structure) expression is an array (or structure) evaluated by expansion of the expression into a collection of scalar expressions and interpretation in an undefined order. Syntactically, an expression consists of a constant, a variable, an expression enclosed in parentheses, an expression preceded by a prefix operator, two expressions connected by an infix operator, or a function reference that returns a value. In a few cases, operands in an expression need not have the same data attributes. If they differ, conversion is performed before the operation.

There are three forms of (sub-)expressions related to the operators contained in them: basic expressions, prefix expressions and infix expressions. A basic expression is a single <reference> or <literal constant>, a prefix expression is a prefix operator with one operand, while an infix expression is an infix operator between two operands. An operand is one of the three forms of <expression>s.

Examples of the expressions:

basic expression	b	x[i]
	s.t	function(x, y)
infix expression	a * b	x + 1
	a * b + c * d	
prefix expression	- n	~ state

4.1 Expressions

Expressions always have values of the same data type and aggregate type, except possibly for differing array-extents, thus they are characterized by these two properties. They are referred to in this specification as: scalar <expression>s, array <expression>s, structure <expression>s, etc.

The data types for an <expression> are described in section 3.2 and the aggregate types are specified in section 3.3.

3.7.3 Default Attributes

Every scalar, array, and fully qualified structure name must have an <attribute set> associated with it in a <declare statement>. The <attribute>s may be declared explicitly, however certain <attribute>s will be assumed as defaults or implied if the <attribute set> is incomplete.

Section 3.5.2 provides the syntax for <attribute set> and <attribute>. This section specifies the defaults and implications for category names in the production of <attribute>. The table below lists the category names, the action or <attribute> if the category name is not explicitly declared, and the <attribute>s implied by the explicit declaration of the category name.

<u>Category Name</u>	<u>Default</u>	<u>Implied Attribute</u>
<storage class>	<automatic>	
<alignment>	compiler selected	
<constant>	variable	
<dimension>	scalar item	
<external>	internal	<static>
<memory>	compiler selected	

There is no defined default for <data type>, <initial>, <precision>, <range>, or <length>.

<u>Attribute</u>	<u>Conflicting Attributes</u>
<aligned>	<unaligned>
<automatic>	<controlled>, <external>, <static>
<bit>	<cardinal>, <character>, <fixed>, <float>, <precision>, <range>
<cardinal>	<bit>, <character>, <fixed>, <float>, <length>, <varying>
<character>	<bit>, <cardinal>, <fixed>, <float>, <precision>, <range>
<constant>	none
<controlled>	<automatic>, <static>
<dimension>	none
<external>	<automatic>
<fixed>	<bit>, <cardinal>, <character>, <float>, <length>, <varying>
<float>	<bit>, <cardinal>, <character>, <fixed>, <length>, <varying>, <range>
<initial>	none
<length>	<cardinal>, <fixed>, <float>
<memory>	none
<precision>	<bit>, <character>, <varying>
<range>	<float>, <bit>, <character>, <varying>
<static>	<automatic>, <controlled>
<unaligned>	<aligned>
<varying>	<cardinal>, <fixed>, <float>, <precision>, <range>

3.7.2 Required Attribute Set Members

A <data type> or <range> attribute is required.

The arithmetic data type attributes <cardinal> and <fixed> require the attribute <precision> or <range>; and the data type <float> requires the attribute <precision>.

The string data type attributes <bit> and <character> require the attribute <length>.

The <constant> attribute requires the attribute <initial>.

Storage allocation for a data item declared with the attribute `<unaligned>` is not required to be aligned on specific boundaries.

3.6.22 Varying

Syntax:

`<varying> ::= VARYING`

An item declared with the attribute `<varying>` represents a string data type whose length may vary after allocation and may be any value from zero to a maximum of the `<length>` specified in the declaration.

3.7 Data Attribute Sets

The description of each `<attribute>` is given independently of other members of the `<attribute set>`. A particular `<attribute>` may conflict with another `<attribute>` such that they cannot belong to the same `<attribute set>`. Alternatively, there are some `<attribute>`s that require a particular `<attribute>` in the same `<attribute set>`. These conflicting or required `<attribute>`s are specified in 3.7.1 and 3.7.2. Defaults exist for certain `<attribute>`s in an `<attribute set>`. The defaults are specified in section 3.7.3.

3.7.1 Conflicting Attribute Set Members

The table below indicates, in the row direction, all `<attribute>`s which conflict with the `<attribute>` in the column on the left.


```
<low value> ::= <cvi expression>  
<high value> ::= <cvi expression>  
<scale> ::= <cvi expression>
```

A name declared with the range attribute specifies an enumeration of fixed-point values and has two uses. One is in declaring the range of values for fixed-point data items, i.e. either <range> or <precision> can be written. In this context the optional parameter <scale> is allowed.

The other use of <range> is to specify a name for a set of values in any context where it is meaningful such as in a <case prefix> or as a subscript to select a cross-section of an array. In this usage the <declared name> does not have <data type>.

The expressions for <low value> and <high value> are integers defining the limits of the closed interval of values.

In the context of precision for fixed-point variables, the <scale> parameter indicates the minimum significance required in computations; the <scale factor> to be used is $\text{CEIL}(3.32*v)$ where v is the value of <scale>.

3.6.20 Static

Syntax:

```
<static> ::= STATIC
```

A variable declared with the attribute <static> is a variable which has storage allocated at or before the time that the variable is first referenced. The variable remains allocated until termination of the <program>.

All <extent> expressions and <initial> attributes for static variables must be constant valued since the variables may be allocated prior to execution of their containing block.

3.6.21 Unaligned

Syntax:

```
<unaligned> ::= UNALIGNED
```

3.6.17 Memory

Syntax:

```

<memory> ::= MEMORY( <memory type> )
<memory type> ::= MDA | MAIN

```

The <memory> attribute specifies the type of memory in which storage for the <declaration component> will be allocated. The <memory type> MDA will cause allocation to be in the multi-dimensional access, or array, memory. The <memory type> MAIN will cause allocation to be in the main, or control memory.

3.6.18 Precision

Syntax:

```

<precision> ::= ( <number of digits>
                  [ , <scale factor> ] )
<number of digits> ::= <cvi expression>
<scale factor> ::=      [+|-] <cvi expression>

```

The <precision> attribute specifies the precision of arithmetic data types. For cardinal or fixed-point data, the <precision> attribute specifies the number of bits sufficient to express the magnitude of the values. For floating-point data, <precision> specifies the number of bits in the magnitude of the mantissa.

The <scale factor> specifies the position of the assumed binary point for fixed-point data. The point is located to the left of the rightmost bit when the <scale factor> is positive and to the right of the rightmost bit when the <scale factor> is negative. If the <scale factor> of a data item with the <fixed> attribute is zero or is omitted, the data item is a fixed-point integer. If <scale factor> is present, the data item must have the attribute <fixed>.

3.6.19 Range

Syntax:

```

<range> ::= RANGE ( <low value>
                   : <high value> [ , <scale> ] )

```

3.6.14 Float

Syntax:

```
<float> ::= FLOAT
```

A data item declared with the attribute <float> represents a floating-point arithmetic value. The <precision> attribute without a "scale factor" is required for floating-point data.

3.6.15 Initial

Syntax:

```
<initial> ::= INITIAL <initial list>
<initial list> ::= ( <initial item>
                    [ ,<initial item>]... )
<initial item> ::= [<factor>]<initial value>
<factor> ::= ( <cvi expression> )
<initial value> ::= [+|-] <literal constant>
```

The <initial> attribute specifies a set of scalar values that are assigned in order to the scalar components of the declared variables. The number of scalar values given in the <initial list> must be equal to the number of scalar components in the <declaration component> with the <initial> attribute. A name that appears in a <parameter list> may not have the <initial> attribute.

3.6.16 Length

Syntax:

```
<length> ::= ( <string length> )
<string length> ::= <cvi expression> | *
```

The <length> attribute specifies the length of string data types. For bit-string data, <length> is the number of contiguous bits in the string. For character-string data, <length> is the number of contiguous characters in the string. The <cvi expression> must be greater than zero.

The <string length> may be an asterisk only for names which appear as parameters in a <parameter list> of a <procedure> in which the name is declared.

The `<dimension>` attribute specifies the extents of the dimensions for data organized as an array or array of structures. If one `<extent>` is given it represents the upper bound of a one dimensional array. If two or three `<extent>`s are given they represent the upper bounds of each dimension of two or three dimensional arrays, respectively. Any `<extent>` written as an `<expression>` must evaluate to a fixed-point integer. The lower bound for all dimensions is one.

The `<extent>` may be an asterisk only for names which appear as parameters in the `<parameter list>` of the `<procedure>` in which the name is declared. In this case all `<extent>`s must be written as asterisks. If the `<extent>` of a parameter is not an asterisk, it must be constant, a `<cvi expression>`.

If the keyword `DIMENSION` is omitted, `<dimension>` must be the first `<attribute>` in the `<attribute set>`.

3.6.11 Entry

An item declared with the attribute `<entry>` represents entry values. Data of type entry and further entry attributes are specified in section 5.3.

3.6.12 External

Syntax:

```
<external> ::= EXTERNAL
```

A name declared with the attribute `<external>` has external scope. The name is known for the scope of the declaration and in all `<block>`s which have the same name declared with the `<external>` attribute.

3.6.13 Fixed

Syntax:

```
<fixed> ::= FIXED
```

A data item declared with the attribute `<fixed>` represents a fixed-point arithmetic value. The `<precision>` attribute is required for fixed-point data.

A name declared with the <constant> attribute is a named constant. Named constants cannot be assigned values during program execution. The <initial> attribute is required to assign a value to the named constant.

A named constant may be used in a <cvi expression> (section 4.3.2).

3.6.8 Controlled

Syntax:

```
<controlled> ::= CONTROLLED
```

A name declared with the attribute <controlled> is a variable which has storage allocated when an <allocate statement> that identifies the controlled variable is executed. The storage for a controlled variable is freed when a <free statement> that identifies the controlled variable is executed.

Only one allocation can exist at any time. See the ALLOCATION builtin-function for the ability to test whether an allocation exists.

3.6.9 Defined

Syntax:

```
<defined attribute> ::= DEFINED <base reference>
<base reference> ::= ( <base reference> )
```

A variable declared with the <defined attribute> is a variable whose generation of storage is identified by the <base reference>. Refer to section 3.3.6 for a discussion of the use of defined variables.

3.6.10 Dimension

Syntax:

```
<dimension> ::= [ DIMENSION ]
               ( <extent> [ ,<extent> [ ,<extent> ] ] )
<extent> ::= <expression> | *
```

3.6.4 Builtin

Syntax:

<builtin> ::= BUILTIN

A name declared with the attribute <builtin> is a builtin-function and must match one of the function names given in section 4.7. This attribute allows redeclaration within an inner procedure block of a name that has been used as a program name in a containing block. The latter is possible since builtin-function names are not reserved words.

3.6.5 Cardinal

Syntax:

<cardinal> ::= CARDINAL

A data item declared with the attribute <cardinal> represents a cardinal arithmetic value. The <precision> or <range> attribute is required for cardinal data.

3.6.6 Character

Syntax:

<character> ::= CHARACTER

A data item declared with the attribute <character> represents a character-string value. The <length> attribute is required for character-string data.

3.6.7 Constant

Syntax:

<constant> ::= CONSTANT

A data item declared with the attribute <aligned> is allocated storage beginning on a boundary determined from the <boundary>.

When <boundary> is BYTE, storage is allocated beginning on byte (8 bit) boundaries. When <boundary> is HALFWORD, storage is allocated beginning on halfword (16 bit) boundaries. When <boundary> is WORD, storage is allocated beginning on word (32 bit) boundaries. When <boundary> is <cvi expression>, the value of 2 raised to the value of <cvi expression> determines the bit boundary for the beginning of allocation. The <cvi expression> must have a value greater than zero. An n-bit boundary occurs at bit positions which are positive, integer multiples of n.

3.6.2 Automatic

Syntax:

<automatic> ::= AUTOMATIC

A name declared in a given <block> scope with the attribute <automatic> is a variable which has storage allocated each time the given <block> is entered. The storage for an automatic variable is freed when the given <block> is deactivated.

The <extent> expressions and <initial> attribute of an automatic variable can contain <expression>s; the values are computed upon block activation. A value is computable on block activation if it can be evaluated without referencing any automatic, controlled, or defined variable declared in the <block>.

3.6.3 Bit

Syntax:

<bit> ::= BIT

A data item declared with the attribute <bit> represents a bit-string value. The <length> attribute is required for bit-string data.

must be followed by a <declaration component> with a <level> greater than one. If a <declaration component> has a level greater than one, it must be a member of a structure.

A given <attribute> can occur only once in an <attribute set>.

3.5.2.1 Declaration of Scalars

A scalar data item is declared by a <declaration component> with a <declared name> that does not have a <dimension> attribute in its <attribute set>, and that is not a main structure or a nested structure.

3.5.2.2 Declaration of Arrays

An array data item is declared by a <declaration component> with a <declared name> that includes a <dimension> attribute in its <attribute set>, and that is not a main structure or a nested structure.

3.5.2.3 Declaration of Structures

A structure data item is declared by a <declaration component> that does not have a <dimension> attribute, and that has a <level>, and is followed by a <declaration component> in the same <declare statement> with a <level> greater than the current <declaration component>.

3.5.2.4 Declaration of Arrays of Structures

An array of structures data item is declared by a <declaration component> that includes a <dimension> attribute in its <attribute set> and is a main or nested structure.

3.6 The Data Attributes

The declaration of a data item includes an <attribute set> to specify properties of the item. The data attributes which may appear in an <attribute set> of a <declare statement> are specified in this section.

3.6.1 Aligned

Syntax:

```
<aligned> ::= ALIGNED ( <boundary> )
<boundary> ::= BYTE | HALFWORD | WORD
               | <cvi expression>
```


All declarations of a name with the <external> attribute must have an identical set of attributes.

3.5.2 The Declare Statement

Syntax:

```

<declare statement> ::= DECLARE <declaration component>
                        [ , <declaration component> ]...;
<declaration component> ::= [<level>] <declared name>
                        [<attribute set>]
                        | [ <level> ]
                          ( <factored name> [ , <factored name> ]... )
                          [<attribute set>]
<level> ::= <decimal integer>
<factored name> ::= <declared name> [ <dimension> ]
<declared name> ::= <identifier>
<attribute set> ::= <attribute> [ <attribute> ]...
<attribute> ::= <data type> | <storage class>
                | <alignment> | <initial>
                | <constant> | <dimension>
                | <external> | <defined attribute>
<data type> ::= <arithmetic> | <string>
                | <entry> | <file>
                | <builtin>
<arithmetic> ::= <float> <precision>
                | { <cardinal> | <fixed> }
                  { <precision> | <range> }
<string> ::= { <bit> | <character> }
              { <length> [ <varying> ]
                | <varying> <length> }
<storage class> ::= <automatic> | <controlled>
                  | <static>
<alignment> ::= <aligned> | <unaligned>
               | <memory>

```

The syntax of all <attribute>s except entry and file is given in section 3.6 and the syntax of entry and file <attribute>s is given in sections 5 and 8.

Factoring may be applied to <declared name>s having <attribute set>s which differ only in the <dimension> attribute.

The optional <level> is used when declaring a structure or array of structures. If a <declaration component> has a level of one, the <declared name> is the name of a main structure. The <declaration component>

3.4.2 Memory Properties

A machine dependent attribute of data is <memory>, which specifies the type of memory where the data item will be allocated. STARAN has two separately addresssed memory systems. The main memory address space is used primarily for storage of scalar data, and small arrays or structures (or those not referenced often). The multi-dimensional access (MDA) memory is used primarily for storage of arrays and structures. Many array operations take place most efficiently on data stored in the MDA memory.

3.4.3 Storage of Arrays

Arrays of data may be stored in the main memory or the STARAN MDA memory. The means of addressing the two memory types are distinct from each other. Thus the organization of data in relation to the memory in which the data is stored may effect performance.

3.4.4 Connected Storage

Arrays are allocated in connected or unconnected storage. A connected array is one in which all elements of the array are adjacent to one another according to a particular means of referencing the array. An unconnected array is one in which elements of the array are separated from one another by other values.

An unconnected array exists as an array of structures or as a result of certain references to portions of the array as described in section 4.2.

3.5 Declarations

Scalar, array, structure, array of structures, entry and file data are given names. The attributes associated with a name are specified by the declaration of the name. All program names must be declared; builtin-function names do not need to be declared.

3.5.1 Scope of a Declaration

The scope of a declared name is the <block> in which the name is declared and all contained <block>s in which the name is not redeclared.

A name can be declared only once in a given <block>. The name is known only for the scope of the declaration unless the <external> attribute is included in the declaration. A name declared with the <external> attribute is known in all <block>s in which the same name is declared with the <external> attribute and in all contained <block>s in which the name is not redeclared.

Example:

```
declare XX(4, 4);
declare Y(4) defined ( XX[lsub, lsub]);
```

The array Y is a four element array whose elements constitute the diagonal of the array XX. A <reference> to Y[K] is equivalent to a <reference> to XX[K, K].

An unsubscripted <reference> to an i-sub defined array is equivalent to a cross-section <reference> in which all <subscript>s are asterisks. In the example above, a <reference> to Y is equivalent to a <reference> to Y[*] which is equivalent to a <reference> to the array formed by the elements XX[1, 1], XX[2, 2], and XX[3, 3]. In this example, Y is always in unconnected storage.

The data type, alignment <attribute>s, and string <length> of the defined array must be identical to the corresponding <attribute>s of the base array. If the defined variable is a structure, the structuring of the defined variable and the base variable must be identical, and the data types, alignment <attribute>s, and extents of all members of the defined variable must be identical to those of its corresponding member in the base structure.

3.4 Data Storage

3.4.1 Storage Classes

The storage class determines the times at which data items are allocated or freed and thus it determines the lifetime for values of the data items. The mutually exclusive storage class attributes are <automatic>, <controlled>, and <static>. Allocation occurs upon <block> entry, execution of the ALLOCATE statement, and at the beginning of <program> execution respectively. The freeing of storage occurs at <block> exit, execution of the corresponding FREE statement, and at the end of <program> execution respectively.

A variable is a member of the storage class corresponding to its declared storage class attribute. Storage for a component of a structure is allocated or freed when its containing main structure is allocated or freed. The storage class attributes are defined further in section 3.6.

4.4.1.2.2 Infix Operations

If both operands are floating-point values, the result is a floating-point value with precision equal to the greater of the precisions of the two operands.

If the operands are fixed-point values, the result type may be either fixed-point or floating-point and depends on both the operator and the converted operands as described by the following:

Notations:

Let N be 63 (maximum floating-point precision, implementation defined, radix is binary).

Let (p,q) be the precision of the first operand, and (r,s) be the precision of the second operand.

In principle the precision and scale are the minimum values that will contain the actual value without loss of significant bits.

Addition or subtraction operator $(+ -)$.

The result is a fixed-point value whose precision is:

$$(\min(N, \max(p-q, r-s) + \max(q,s) + 1), \max(q,s))$$

The value of the result is the algebraic sum or the difference of the two operands.

Multiplication operator $(*)$.

The result is a fixed-point value whose precision is:

$$(\min(N, p+r+1), q+s)$$

The value of the result is the algebraic product of the two operands.

Division operator $(/)$.

The result is a fixed-point value whose precision is given by:

$$(N, N-p+q-s)$$

The value of the result is the algebraic quotient of the first operand divided by the second. If the quotient exceeds the precision of the result, the least significant digits of the quotient are truncated to form the result. Note that the result always has the maximum precision

The precision of the two operands may differ without requiring any conversion. If a conversion built-in function is used to overcome a difference in type, the precision of the converted operand is given by the rules in section 4.5.

Example:

```
declare A character(5), B fixed (31, 0);
```

```
... = fixed(A) + B;
```

In this example, A is converted to a fixed-point quantity of precision(16, 0); large enough to hold any 5 digit decimal integer. The target type for the addition is fixed-point. The operand B is not converted because it already has the target attribute <fixed>.

4.4.1.2 Arithmetic Operator Results

The operands are converted (possibly using built-in functions) and evaluated, then the operation is performed. The result is an arithmetic value whose type and precision are determined by the converted operands and the operator as described in the following sections.

The precision rules of "fixed-point" operations are such that no high order digits (bits) of the true arithmetic result are lost. Unless the operation is division or the result precision has reached the limits of the machine-system(N), no low order digits of the true arithmetic result are lost. In the latter case, the precision rules given below indicate exactly when low order digits are lost.

A floating-point result of precision(p), always binary, contains the most significant (n) bits of the true arithmetic result, where n is implementation defined, is greater than or equal to p, and is a maximum of 71.

When the final result of the evaluation of an <expression> is assigned to a variable or to a generation of storage to be passed (by value) as an argument, as many as n significant digits are stored if the target is unpacked(aligned) and of significant precision, but p significant digits are stored if the target is packed(unaligned). In the truncation case, excess low order digits are truncated.

4.4.1.2.1 Prefix Operations

The two prefix operators, plus and minus, produce a result having the type and precision of the single operand. The value of the result of a plus operator is the value of the operand. The value of the result of a minus operator is the algebraic negative of the operand value.

program. It can be referenced in any context that requires constant-valued integers.

Syntax:

`<cvi expression> ::= <expression>`

A `<cvi expression>` can be a name declared with only the `<range>` attribute; no `<data type>` is needed. In such a declaration it then represents the set of integer values defined with the name. Alternatively, the `<cvi expression>` can be any expression which is made up of only constant-valued integer operands.

A `<cvi expression>` can be referenced in any context that requires constant-valued integers.

4.4 Operators

4.4.1 Arithmetic Operators

The prefix arithmetic operators are:

- + plus
- minus

The infix arithmetic operators are:

- + add
- subtract
- * multiply
- / divide
- ** exponentiation

4.4.1.1 Arithmetic Operands Operator

The operands for arithmetic operators must both be arithmetic. The conversion rules are given in section 4.5, and the target attribute set for the conversions is:

1. If one operand has the attribute `<fixed>` and the other is `<cardinal>`, the latter operand is converted to `<fixed>` precision `(r,0)` where `r` is the maximum of the two precisions.
2. In all other cases, the operands must be explicitly converted to a common arithmetic type using a built-in function.

Order of Evaluation.

4.3.1 General Expressions

Syntax:

```

<expression> ::= <expression one>
               | <expression> | <expression one>

<expression one> ::= <expression two>
                   | <expression one> & <expression two>

<expression two> ::= <expression three>
                   | <expression three> <relational> <expression three>

<relational> ::= = | ~= | ≤ | ~≤ | ≤= | ≥ | ~≥ | ≥=

<expression three> ::= <expression four>
                    | <expression three> || <expression four>

<expression four> ::= <expression five>
                    | <expression four> { + | - } <expression five>

<expression five> ::= <expression six>
                    | <expression five> { * | / } <expression six>

<expression six> ::= <basic expression>
                   | <simple expression>
                   | <parenthesized expression> | <expression seven>

<expression seven> ::= { <basic expression>
                       | <parenthesized expression> }
                     ** <expression six>

<simple expression> ::= { + | - | ~ } <expression six>

<parenthesized expression> ::= ( <expression> )

<basic expression> ::= <reference>
                     | <literal constant> | <i-sub>

```

4.3.2 Constant-valued Expressions

A constant-valued expression is any expression which involves only constant integer operands and can be evaluated before any execution of the

the parameter is associated with an argument, and holds only for the time the parameter remains associated with the argument.

If a parameter is declared with (only) constant extents, the corresponding argument must have identical constant extents to match the parameter.

In determining whether an argument is to be passed by-value or by-reference, an asterisk extent is considered to "match" any extent of the argument (see 4.2.8.2). A dimensioned parameter that could correspond to an array argument in unconnected storage must be declared with asterisk bounds.

4.2.9 Function Reference Reducibility

An entry is a reducible function under the following circumstances:

- Each invocation of an entry invokes only reducible functions.
- Each invocation of an entry returns a value that depends only on the values of the arguments passed to that invocation.
- Each invocation of an entry produces no side-effects.

Any entry that is not reducible is irreducible. Note that a side-effect is any change in the value of any item outside of the invoked entry or any of its dynamic descendents.

The order of evaluation of <reference>s to irreducible functions is not defined during the evaluation of <expression>s, but all the function <reference>s are evaluated. A <reference> to a reducible function may be evaluated before the statement in which it is written is executed, but a <reference> to an irreducible entry is always evaluated during execution of the statement in which it is written.

See section 4.1 for order of evaluation in <expression>s and section 5.3 for the definition of the <reducible attribute> and <irreducible attribute>.

4.3 Formal Syntax of Expressions

This section defines the expressions, and the order of evaluation of operators contained in <expression>s (thus the precedence). The evaluation of operands in <basic expression>s may differ from the order for operations as expressed by these syntax rules according to the rules of section 4.1.4,

4.2.8.2 Matching Attributes for By-reference Passing

In procedure invocations, the attributes of a <reference> written as an argument must match those of the corresponding parameter in order to achieve passing the argument by-reference. If the attributes do not match, the arguments are either passed by-value or the program is in error.

The following rules also apply:

If a parameter is an aggregate (structure or array), it matches an aggregate argument with elements that match the corresponding elements of the parameter. It also matches a scalar argument if the latter can be promoted as specified in section 4.6.

If a parameter is a scalar, the <fixed>, <float>, <bit>, <character>, <entry>, <file>, <varying>, <aligned>, <unaligned>, and <memory> attributes must match.

If a parameter is arithmetic, the <precision> attribute of the parameter must be * or have the same value as the argument <precision>. Implicit conversion is allowed between types <fixed> and <cardinal> as defined in section 4.5.

If a parameter is an entry variable, it matches an argument which is an expression with type <entry>. The <parameter description list> is not matched.

If the parameter is an array, the <dimension> attribute of the parameter must have all *-extents or have the same values as the array argument. The array parameter can match a scalar argument if the latter can be promoted as specified in section 4.6. An array in unconnected storage (cross-section or "&" subscript reference) cannot be passed as an argument to an array parameter declared with constant extents.

4.2.8.3 Promotion and Conversion of Arguments

As specified in the previous section, the evaluation of an argument to be passed by-value may include promotion and/or conversion. If the only defined promotions or conversions will not allow the argument to conform to the parameter, then the program is invalid and in error. Refer to sections 4.5 and 4.6 for definition of conversions and promotions respectively.

4.2.8.4 Extents of Parameters

An array or string parameter may be declared with either <cvi expression> (constant) or asterisk extents. For a parameter declared with asterisk extents, the unspecified extents are given the values of the corresponding argument extents; this assignment of extents occurs each time

4.2.7 Built-in Function References

In addition to function references to user written procedures, a function reference can invoke one of the set of built-in functions. This set is an intrinsic part of the language and provides operations that either cannot be expressed or are not usually expressed as infix or prefix operations. The identifiers used for the built-in function names are not reserved. Any of the identifiers can be declared for other items and, if necessary, redeclared as built-in functions in an inner block by use of the <builtin> attribute. A built-in function name does not have an entry-value and must not be written in a context that requires an entry value. The complete list of functions and their specification is in section 4.7.

4.2.8 Parameters and Arguments

As shown in the syntax, an argument is an <expression> in the <argument list> of a <call statement> or <function reference>. A parameter is a name declared in the procedure and used by the latter to reference an argument. The correspondance between an argument and a parameter is by position in the respective lists and exists until the block activation that established the correspondance is ended by a return.

The scope of a parameter is internal to the <block> in which the name appears as a parameter, and the generation of storage associated with a parameter is supplied by its corresponding argument. A parameter cannot have an <initial> attribute.

A reference to a parameter is in error if there is no argument associated with the parameter. This erroneous reference could only occur in a <procedure> containing one or more <routine statement>s.

4.2.8.1 Argument Passing By-value or By-reference

Arguments may be passed in two manners, by-value and by-reference. For passing by-value, the argument is evaluated and assigned to a new generation of storage in the invoking <procedure>. The new generation of storage is then associated with the parameter. For passing by-reference, the argument's generation of storage is associated with the parameter; the argument and parameter share one generation of storage. If an argument is passed by-value, any assignment of values to the corresponding parameter do not access or change the value of the argument.

In order to be passed by-reference, an argument must be both readable and writeable; thus a <reference> enclosed in parentheses as a sub-expression, a <literal constant>, or a <reference> to a variable with the attribute <constant> is always passed by-value.

- The reference is a complete qualification of only one declaration of the name.

The reference in either case is to the declared name. In any other case the reference is ambiguous and in error.

The form of any <subscript> lists or <argument list>s does not affect the resolution of <reference>s.

4.2.6 Function Reference

The syntax of a <function reference> differs from that of a <subscripted reference>.

Syntax:

```
<function reference> ::= <entry reference>
                        [ <argument list> ]
<entry reference> ::= <reference>
<argument list> ::= ( [ <arguments> ] )
<arguments> ::= <expression> [, <expression> ]...
```

The <entry reference> is required to yield a scalar entry value. A <function reference> is distinguished from a <reference> to an entry value by the presence of an <argument list>. The <argument list> must be empty if the entry has no

parameters. The evaluation of a <function reference> results in the invocation of an entry value. The value of the <function reference> is the value returned by the invoked entry.

Examples:

```
declare F_NO_ARGS entry() returns(fixed(15, 0));
declare F1_ARG entry (fixed(15, 0)) returns (bit(1));
```

A <reference> to F_NO_ARGS is not a <function reference>, but is a <reference> to the entry of F_NO_ARGS. A <function reference> to the value returned by the invocation of F_NO_ARGS would be written as F_NO_ARGS().

A <reference> to the entry value of F1_ARG is written as F1_ARG and a <function reference> to F1_ARG is written as F1_ARG(K).

Example:

```
declare 1 LIST(3),
        2 NEXT ... ,
        2 DATA(4);
```

The items LIST and NEXT are one-dimensional arrays and a <subscripted reference> must contain one <subscript>. A <subscripted reference> to DATA or LIST.DATA must have two <subscript>s since it has a dimension of two.

Cross-sections of these arrays may be referenced as follows:

Example:

```
declare 1 LIST(3),
        2 NEXT cardinal(16) ,
        2 DATA(4) bit(8);

declare 1 TAB,
        2 ELEMENTS(5) bit(8);
```

The following are all valid cross-section <reference>s:

LIST[K].DATA[*]	LIST[*].DATA[*]	LIST[*].NEXT
LIST[*]	TAB.ELEMENTS[*]	

The following is an invalid reference:

```
LIST.NEXT[ ... ]
```

4.2.5 Qualified Reference Resolution

A qualified reference to a name applies to the valid declaration of the structure identifier declared in the innermost block containing the reference. A valid declaration is one of a structure <member> in which some or all of the containing structures have the same names and (hierarchical) order as the <containing reference>s of the subject reference.

Within this block of declaration, the reference is unambiguous if either of the following holds:

- The reference is a valid qualification for exactly one declaration of the name.

data and to section 3.6 for a description of structure declarations.

Syntax:

```

<qualified reference> ::= <containing reference>
    [ . <containing reference> ]...
    . <member reference>
<containing reference> ::= <simple reference>
    | <subscripted reference>
<member reference> ::= <simple reference>
    | <subscripted reference>
    | <function reference>

```

Examples:

K.L.M

R.S.T[2, M]

NODE.ELEMENT

In a structure reference, the rightmost <reference> is the <member reference> and it identifies the innermost item being referenced. It is contained within the structure identified by the immediately preceding <containing reference>, which in turn is contained within the structure identified by the immediately preceding <containing reference>, etc.

The structure reference is fully qualified if it has a <containing reference> corresponding to each of its containing structures. In order to be in a fully qualified <reference>, a name declared at level n in a substructure must have n-1 <containing reference>s. A <simple reference> or simple <subscripted reference> to a name declared by a level-one declaration is considered to be a fully qualified <reference> to that name.

A fully qualified <reference> is never ambiguous. A rightmost <reference> is partially qualified if it has fewer <containing reference>s than it has containing structures; a partially qualified reference can be ambiguous.

The number of <subscript>s in a reference must always be the same as the declared dimension of the referenced name including all dimensions inherited from containing structure levels.

<identifier> [& <expression>]

where <expression> is the value obtained at the time the IFARPAY was executed. In the context of IFARRAY...ELSE...& it is equivalent to

<identifier> [~(&<expression>)]

4.2.3.3 Selection by Subscript Range

Form: <first element> : <last element>

<first element> ::= <expression>
<last element> ::= <expression>

The elements selected are those in the given dimension with indices in the range of the value of the expression <first element> through the value of the expression <last element>.

The resultant reference is an unconnected array reference if any subscript for a higher numbered dimension (a subscript toward the left) is a <selector>. It is a connected array reference if each and every subscript, if any, for higher numbered dimensions is an expression.

Example:

```
declare (A(20), B(20), C(22)) cardinal(16);

ifarray B[*] < A[*] then
  B[&] = A[&] + C[3 : dim(C) ];
endif;
```

For each i-th element of B which is less than the corresponding (i-th) element of A, assign the sum of A[i] and C[i+2].

4.2.4 Structure Qualification

A structure is a hierarchical collection of scalar variables, arrays, and structures. These do not need to be of the same data type nor have the same attributes. Because of the hierarchy, the name of a structure member (item) can have a scope that overlaps another declaration of the name.

Structure qualification provides the facility to resolve ambiguity of <reference>s to these names. In structural qualification, the <reference> is qualified by <containing reference>s to one or more of its declared containing structures. Refer to section 3.3 for a discussion of structure

There are 3 forms of array selection qualification:

1. & <selection qualifier>
2. &
3. <first element> : <last element>
where each of these is an <expression>

All three are means of selecting a portion of an array in the dimension corresponding to the position in which the <subscript> appears in the <subscripted reference>. The portion of the array being selected is an unconnected cross-section reference. The number of dimensions of the reference is not reduced by these subscript forms.

4.2.3.1 Array Slice Selection

Form: & <selection qualifier>

The <selection qualifier> is a <reference> and must be an array of bit-string values (usually "bit(1)"). If the extent of the <selection qualifier> vector is not equal to the corresponding dimension of the array reference, a copy of its value is extended to the extent of the latter dimension value.

A list of values for the subscript is developed as follows:

If the bit string in element *i* of the bit-string array is "true" (not all zero bits), the corresponding subscript has the value *i* added to the list. If the bit string in element *i* of the bit-string array is "false" (all zero bits), the subscript value *i* is not used in forming the list.

The resulting list of subscript values is used to select array elements from the array; the number in a given dimension is equal to the number of "true" values in the <selection qualifier> bit-string array.

If the array qualified by this form has only one dimension, then the reference makes a vector referred to as a selection by bit-vector.

4.2.3.2 Special Selection in IFARRAY Context

Form: &

This <selector> form is allowed in the context of an <ifarray statement> to select array elements based on the value of the <expression> in the innermost containing <ifarray clause>. The reference

<identifier> [&]

in the context of IFARRAY...THEN...& is equivalent in meaning to

asterisk. As for all array references, the number of subscripts in the list must be equal to the dimension declared with the array name. If the j -th subscript is an asterisk, the cross-section of the array includes all elements covered by varying the j -th subscript in the interval closed by its declared bounds. The dimensionality of the cross-section reference is equal to the number of asterisks, k , in the subscript list.

If all <subscript>s are asterisks, then this reference is equivalent to the name written without any subscripts; it is a reference to the entire array.

A cross-section may be used anywhere that a reference to an array of dimension k is required. In this document the word "array" should be interpreted to include array values of cross-section <reference>s.

A reference to a cross-section of an array may be a reference to elements which are not in contiguous storage; the elements are separated by data items that are not part of the cross-section. Such unconnected arrays cannot be passed by-reference as arguments to array parameters unless the parameters have a declaration with an asterisk dimensions. In all other contexts, a cross-section <reference> can be used wherever an array <reference> could be. See section 3.4.3 for the discussion of connected arrays.

Example:

```
declare I cardinal(16) ;
declare (K(8, 8), L(8, 4), M(8) ) bit(16) ;
```

```
K[I, *] = L[* , 2] + M[*];
```

The first assignment statement computes a vector of 8 values by adding the values of the second column of L to the values of M and then assigning the sums to the I -th row of K . Note that a <reference> to $M[*]$ is equivalent to a <reference> to M .

4.2.3 Selection Qualification

A <subscripted reference> containing one or more <selection qualifier>s as subscripts is a cross-section reference to the array. It is a reference to an unconnected array of elements selected by the one bits in the bit-string <selection qualifier> from the array specified by the <reference>.

Any <reference> contained in a declaration of a name X is resolved in the <block> that immediately contains the declaration of X, and is evaluated when X is referenced or allocated. The <reference> is evaluated as if it were referenced in the <block> that immediately contains the <reference> to X, or in the <block> that caused the allocation of X.

4.2.1 Simple Reference

Syntax:

```
<simple reference> ::= <identifier>
```

The reference is to any <variable>; it may be a scalar, an array, or a structure (including array of structures). A reference to a data item other than a scalar is known as an aggregate value if the context requires its value.

4.2.2 Array Reference

Syntax:

```
<subscripted reference> ::=
    <array reference> [ <subscript> ]
    | <array reference>
      [ <subscript> { , <subscript> }... ]
```

```
<array reference> ::= <simple reference>
```

```
<subscript> ::= <expression> | *
              | <selector>
```

```
<selector> ::= & | & <selection qualifier>
              | <first element> : <last element>
```

```
<selection qualifier> ::= <reference>
```

The number of <subscript> expressions must be the same as the number of dimensions in the <dimension> attribute of the declaration of the name (see 3.6). The <expression>s used as a <subscript> are scalar values used as type <fixed-point> or <cardinal> with integer values. Refer to section 3.3 for a discussion of array data.

Cross-sections are an extension of the <subscripted reference>. A cross-section of an array, when referenced by the array name followed by a list of subscripts, is denoted when at least one of the subscripts is an

Syntax:

```

<reference> ::= <simple reference>
              | <subscripted reference>
              | <selected reference>
              | <qualified reference>
              | <function reference>

```

The semantics of a <reference> depends upon <reference>, the <attribute>s of the declared name referred to, and upon the context and form in which the <reference> occurs.

Evaluation of a <reference> that identifies a name without the <constant> attribute either denotes a generation of storage of the variable, the current value stored in the generation of storage of the variable, or an identification of the variable's declaration.

A <reference> yields a value unless it occurs in one of the two forms of contexts that follow:

In these contexts, a <reference> to a variable identifies only the declaration of the variable:

- <allocate reference> for an <allocate statement>.
- argument to the built-in function "ALLOCATE".

In several contexts, the <reference> produces the complete description of its generation of storage, which must be allocated, but not the value:

- assignment statement <target>.
- <index> in an <iterative loop> heading (DO-statement).
- free statement <reference>.
- argument, passed by-reference, in a <call statement> or <function reference>.
- pseudo-variable <reference>.
- <reference> in an <in option>, <into option>, <from option>, or <keyto option>.

In all other contexts, a <reference> accesses the value of the variable. Both the generation of storage must be allocated, and a value must have been previously assigned; otherwise the program is in error. The components of a <reference> are evaluated in an unspecified order, and any program that depends on a certain order is in error.

cannot return.

Any <on unit> signalled during <expression> evaluation cannot access variables that are assigned values by the interrupted expression or containing statement. Similarly, the <on unit> cannot assign new values to variables used by the interrupted statement with the expectation that the new value will be used upon normal return from the <on unit>.

Example:

```

on zerodivide ;
    :
    :
endon;
:
:
D = 0;
X = 1/D + 1/D;

```

In this example:

1. the value of X is not defined upon entry to the <on unit>.
2. if the <on unit> does a normal return the result of the program is undefined and in error, regardless of whether or not the <on unit> assigned a new value to D.
3. the number of times the zerodivide condition is signalled is not defined, but it is signalled at least once.

4.2 References

References provide the means of access to all names declared in a <program unit>. The value and/or storage of a variable, the value of a named constant, or the value returned by a function are all represented in the text of a <program unit> by a <reference> to a declared name. Refer to Section 3.5 for a discussion of declarations and the scope of a declaration.

A <reference> must be associated with a single declaration. A <reference> is resolved by determining the declaration to which it refers; resolution is described in section 4.2.5. A <reference> must be associated with a single declaration. A <reference> is evaluated by locating the generation of storage or value represented by the declared name. A <reference> is resolved by the compiler and is evaluated during program execution.

The <expression> is the quantity to be converted to a character-string of minimum length to represent the character equivalent value of the argument.

4.7.1.15 VERIFY

The VERIFY built-in function will examine two given strings to verify that each character in the first string is present in the second string.

Format:

```
VERIFY( <expression>, <expression> )
```

Both <expression>s must be either of type character-string, or of type bit-string. The first <expression> is the source string and is compared against the second <expression> to see if each element of the first exists in the second. If each character of the source expression exists in the second, this function returns a value of zero. If an examined character in the source expression is not present, the index of that character is returned.

4.7.2 Arithmetic Built-in Functions

4.7.2.1 ABS

The ABS built-in function will determine the absolute value of a given quantity and return it to the point of reference.

Format:

```
ABS( <expression> )
```

The <expression> argument must be arithmetic, and the value returned by ABS is the absolute value of <expression>. If <expression> is an array or structure, then the absolute value of each member is returned.

4.7.2.2 ADD

The ADD built-in function permits one to control the precision of an addition operation.

Format:

```
ADD( <expression>, <expression>, <result precision>  
    [, <result scale> ] )
```

The first two <expression> terms are arithmetic in type and are added together according to the precision and scale specified by the last two terms. The <result precision> specifies the precision, and the <result scale>, when present, specifies the scale. Both <result precision> and <result scale> are defined to be <cvi expression>s. If the type of the

Format:

SOME(<expression>)

the argument <expression> is an array of bit-strings. If the elements are not bit-strings, they are converted to bit-strings. The scalar value returned by this function is a bit-string of length one and whose bit value is determined by the following rule: If any bit of any element in <expression> is B'1', then the result is B'1'; otherwise, the result is B'0'.

4.7.1.12 STRING

This built-in function will concatenate all of the elements of an aggregate variable into a single string element.

Format:

STRING(<reference>)

The <reference> is that of the aggregate variable whose elements are to be converted to their common string type (character, unless all of the elements of <reference> are of type bit-string).

4.7.1.13 SUBSTR

The SUBSTR built-in function provides access to part of a specified string.

Format:

SUBSTR(<expression>, <expression> [, <expression>])

The first <expression> term represents the string from which the substring will be extracted. The second occurrence of <expression> is the index of the starting point in <expression> of the desired substring. The third occurrence of <expression>, if present, specifies the length (number of bits or characters) of the desired substring. If the third expression is not present, the substring extends through the end of the string <expression>.

4.7.1.14 VARCHAR

The VARCHAR built-in function will convert a given value to a varying character-string.

Format:

VARCHAR(<expression>)

The evaluation of <expression> specifies the length of the string to be formed.

4.7.1.8 INDEX

The INDEX built-in function determines if a specified bit-string or character-string occurs within a specific string and, if so, its position within the string.

Format:

INDEX(<expression>, <expression>)

The evaluation of both <expression>s must be of either type bit-string or of type character-string. The first <expression> is the string to be searched, and the second <expression> is the string configuration to be searched for. The resultant value is zero or the position of the beginning of the first occurrence of the second string <expression>.

4.7.1.9 LENGTH

The LENGTH built-in function finds the length of a given string value. The value of the function is the number of bits in a bit-string or the number of characters in a character-string.

Format:

LENGTH(<expression>)

The <expression> represents the string whose length is determined.

4.7.1.10 LOW

The LOW built-in function forms a character-string of a particular length consisting of the lowest character in the collating sequence.

Format:

LOW(<expression>)

The <expression> term specifies the length of the string to be formed.

4.7.1.11 SOME

The SOME built-in function tests the bits of a given bit-string array and returns the result, in the form of an element bit-string, to the point of invocation. The element bit-string indicates whether or not at least one of the bits of the given array elements is set to B'1'.

Format:

COLLATE

The value returned by this function is a character-string of length 256. The characters are the 256 characters of the 8-bit ASCII code in the standard collating sequence order.

4.7.1.5 COPY

The COPY built-in function will form a string consisting of a specified number of repetitions of a given string.

Format:

COPY(<expression>, <expression>)

The evaluation of the first <expression> is of type string, and is the basis from which the new string will be formed. The second <expression> specifies the number of repetitions desired. The length of the result is the length of the first <expression> string times the value of the second <expression>.

4.7.1.6 EVERY

The EVERY built-in function tests all bits of a given bit-string array and returns the result, in the form of an element bit-string, to the point of invocation. The element bit-string indicates whether or not the bits of given array elements are all ones.

Format:

EVERY(<expression>)

The argument <expression> is an array of bit-strings. If the elements are not bit strings, they are converted to bit-strings. The scalar value returned by this function is a bit-string of length one whose bit value is determined by the following rule: If the bits of all of the elements in <expression> are B'1', then the result is B'1'; otherwise, the result is B'0'.

4.7.1.7 HIGH

The HIGH built-in function will form a character string of a given length from the highest character in the collating sequence.

Format:

HIGH(<expression>)

The FIXED <expression> value is converted to decimal radix representation. The precision of the <expression> is divided by 3.32 and the smallest integer greater than or equal to the quotient is obtained. The latter is the derived length and is used as the number of decimal digits for the converted value. The derived length is then increased by two to allow for sign and a decimal point.

(1) If the length of the source <expression> is equal to the specified <result length> value, or the <result length> term is omitted, then the result is the converted value of <expression>.

(2) If the length of the source <expression> is less than the specified <result length> value, the result is padded on the right with blanks to the specified length.

(3) If the length of the source <expression> is greater than the specified <result length> value, the STRINGSIZE condition is raised. For a normal return, the result is truncated on the right to the specified length.

If the source is FLOAT:

The FLOAT <expression> value is converted to decimal scientific notation according to the definition of a <floating-point constant>. The precision of the <expression> is divided by 3.32 and the smallest integer greater than or equal to the quotient is obtained. The latter is used as the number of decimal digits to represent the mantissa portion of the number; additional characters are used as specified for a <floating-point constant>. The exponent of the latter will contain two digits.

(1) If the length of the source <expression> is equal to the specified <result length> value, or the <result length> term is omitted, then the result is the converted value of <expression>.

(2) If the length of the source <expression> is less than the specified <result length> value, the result is padded on the right with blanks to the specified length.

(3) If the length of the source <expression> is greater than the specified <result length> value, the STRINGSIZE condition is raised. For a normal return, the result is truncated on the right to the specified length.

4.7.1.4 COLLATE

The COLLATE built-in function returns the full ASCII character set in collating sequence order.

(2) If the length of the source <expression> is less than the specified <result length> value, then <expression> is padded on the right with blanks to the specified length.

(3) If the length of the source <expression> is greater than the specified <result length> value, the STRINGSIZE condition is raised. For a normal return, <expression> is truncated on the right to the specified length.

If the source is BIT:

(1) If the source <expression> is null, the result is null.

(2) If the precision of the source <expression> is equal to the specified <result length> value, or if the <result length> term is omitted, then the result is produced containing 0 and 1 characters in the same pattern as the zero and one bits of the source <expression> value.

(3) If the precision of the source <expression> is less than the specified <result length> value, then <expression> is padded on the right with blanks to the specified length.

(4) If the precision of the source <expression> is greater than the specified <result length> value, the STRINGSIZE condition is raised. For a normal return, <expression> will be truncated on the right to the specified length.

If the source is CARDINAL:

The CARDINAL <expression> value is converted to decimal radix representation. The precision of the <expression> is divided by 3.32 and the smallest integer greater than or equal to the quotient is obtained. The latter is used as the length in terms of the number of decimal digits for the converted value.

(1) If the length of the source <expression> is equal to the specified <result length> value, or the <result length> term is omitted, then the result is the converted value of <expression>.

(2) If the length of the source <expression> is less than the specified <result length> value, the result is padded on the right with blanks to the specified length.

(3) If the length of the source <expression> is greater than the specified <result length> value, the STRINGSIZE condition is raised. For a normal return, the result is truncated on the right to the specified length.

If the source is FIXED:

(3) If the precision of the source <expression> is greater than the specified <result length> value, raise the STRINGSIZE condition. For a normal return, <expression> is converted as in case (1) and is truncated on the right to the specified length.

4.7.1.2 BOOL

The BOOL built-in function will produce a bit-string whose bit representation is the result of a given Boolean operation on two given bit-strings.

Format:

```
BOOL( <expression>, <expression>, <expression> )
```

The evaluation of the three <expression>s must be of type bit-string. The first and second <expression>s are combined according to the Boolean operation specified by the third <expression>. The third <expression> term is a scalar bit-string of length four (4), selecting one of the sixteen functions. The first two <expression>s may be arrays or structures. If so, they obey the rules governing conformance and promotion (4.6) and the BOOL function is performed upon each corresponding pair of elements.

4.7.1.3 CHARACTER

The CHARACTER built-in function converts a given value to a character-string. This function allows the specification of the length of the character-string result.

Format:

```
CHARACTER( <expression> [, <result length> ] )
```

The <expression> term is the quantity to be converted to a character-string. The <result length>, when specified, is a cvi expression or variable <expression> denoting the length of the character-string result. If <result length> is not constant, the resultant character string has the attribute VARYING. If it is not present, the length is determined by the conversion rules according to the attributes of the first <expression>.

The rules for conversion to CHARACTER depend on the type of the source data. The rules follow:

If the source is CHARACTER:

(1) If the length of the source <expression> is equal to the specified <result length> value, or if the <result length> term is omitted, then the result is the value of the source <expression>.

zeroes to the specified length.

(4) If the length of the source <expression> is greater than the specified <result length> value, raise the STRINGSIZE condition. For a normal return, <expression> will be truncated on the right to the specified length.

If the source is CHARACTER:

(1) If the source <expression> is the null string, the result is null.

(2) If the source <expression> contains only 0 or 1 characters and its length is equal to the specified <result length> value, then the converted result is obtained as follows: The bit string result is produced containing the same pattern of zero and one bits as there are 0 and 1 characters in the source <expression>.

(3) If <expression> contains only 0 or 1 characters and its length is less than the specified length, then the result produced as in case (2) is padded on the right with zeroes to the specified length.

(4) If <expression> contains only 0 or 1 characters, and its length is greater than the specified <result length> value, then the result produced as in case (2) is truncated on the right to the specified length.

(5) Else, raise the CONVERSION condition for a character which is neither a 0 or 1.

If the source is FIXED:

(1) The conversion is equivalent to converting the source <expression> to CARDINAL and then to BIT.

If the source is FLOAT:

(1) The conversion is equivalent to converting the source <expression> to FIXED, then to CARDINAL, and finally to BIT.

If the source is CARDINAL:

(1) If the precision of the source <expression> matches the specified <result length> value, or the <result length> is omitted, then the result is obtained as follows: The binary values of the cardinal's internal representation are used as the bits of the resultant bit string.

(2) If the precision of the source <expression> is less than the specified <result length> value, then the converted result obtained as in case (1) is padded on the right with zeroes to the specified length.

COS	LOG2	SIN	SQRT
TAN			

4. Array Built-in Functions.

DIM	SELECT	SUM
-----	--------	-----

5. Miscellaneous Built-in Functions.

ALLOCATION

The asterisk (*) notation on certain built-in functions denotes those built-in functions that are essentially conversion functions.

4.7.1 String Built-in Functions

The string built-in functions provide the user with the ability to manipulate string values in a wide variety of ways.

4.7.1.1 BIT

The BIT built-in function allows one to control the result of a bit-string conversion, or to force conversion to bit-string.

Format:

BIT(<expression> [, <result length>])

The quantity to be converted to bit-string is the source <expression> , and the <result length>, when specified, gives the length of the result. The <result length> term is defined to be a <cvi expression>. If the second argument is omitted, the length is the <length> or <precision> of the first argument.

If the source <expression> is an array or structure, then the conversion of each member is returned.

The rules for conversion to BIT depend on the type of source data. The rules follow:

If the source is BIT:

- (1) If the source <expression> is the null string the result is null.
- (2) If the length of the source <expression> matches the <result length> value, or the latter is omitted, the result value is the value of the source <expression>.

- (3) If the length of the source <expression> is less than the specified <result length> value, the converted result is padded on the right with

2. Scalars are promoted to structures by forming a structure whose members each have a copy of the scalar value.

3. Scalars are promoted to arrays of structures by forming an array of structures whose scalar components each have a copy of the scalar value.

4. Structures are promoted to arrays of structures by forming an array of structures whose array elements each have a copy of the structure value.

A data item which is an array cannot be promoted to another aggregate type.

Example:

```
declare ( A[5], B[4], C[2, 2] ) fixed(31, 0);
```

In this example, there are no valid promotions between any pairs chosen from A, B, and C.

4.7 Built-in Functions

These functions are defined as part of the language and are shown in five classifications.

1. String Built-in Functions.

*BIT	BOOL	*CHARACTER	COLLATE
COPY	EVERY	HIGH	INDEX
LENGTH	LOW	SOME	STRING
SUBSTR	*VARCHAR	VERIFY	

2. Arithmetic Built-in Functions.

AES	ADD	*CARDINAL	CEIL
DIVIDE	*FIXED	*FLOAT	FLOOR
INDEX_FIRST	INDEX_MAX		
INDEX_MIN	INTERVAL_TEST	MAX	
MIN	MOD	MULTIPLY	PRECISION
SUBTRACT	TRUNC		

4. Mathematical Built-in Functions.

```
call SUBR(x);
```

In this example, the scalar argument *x* is promoted to a two-dimensional array of one element whose value is *x*.

4.6.2 Types of Promotion

The language defines promotion for these four combinations of data organization:

```
from scalar to array
from scalar to structure
from scalar to array of structures
from structure to array of structures
```

The language does not define promotion from arrays to any aggregate type, nor does it define promotion from structures to any aggregate type other than arrays of structures.

The word promotion indicates a ranking of aggregate types, and the promotion of the operands of infix operators utilizes this ranking. The aggregate types are ranked as follows:

```
highest  array of structures
equal    array or structure
lowest   scalar
```

There are four rules for promotion and they apply as shown here:

TO: FROM:	array of structures	structure	array	scalar
-----	-----	-----	-----	-----
array of structures	-	none	none	none
structure	(4)	-	none	none
array	none	none	-	none
scalar	(3)	(2)	(1)	-

4.6.3 Promotion Rules

1. Scalars are promoted to arrays by forming an array whose elements each have a copy of the scalar value.

4.6 Conformance of Aggregate Types

An aggregate type refers to the dimensionality, array-extents, and structuring of a set of (scalar) values (defined in section 3.3). A value conforms to an aggregate type if it has the dimensionality, array-extents and structuring specified by the aggregate type. When a value does not conform to the aggregate type required by the context in which the value appears, it is promoted, if possible, as required by the context. If the promotion required is not defined, the program is in error. The types of promotions are defined in section 4.6.2.

4.6.1 Contexts That Force Promotion

Promotion is attempted in the following contexts:

1. The aggregate type of the value of the <expression> of an <assignment statement> may be promoted to the aggregate type of the <target> of the <assignment statement>.
2. The value of an argument of a <function reference> or <call statement> may be promoted to the aggregate type of the corresponding <parameter descriptor> of the entry declaration.
3. The two operands of infix operators are promoted to the higher of the aggregate types.
4. The value of a <return value> is promoted to conform to the aggregate type specified by the <returns attribute> of the <entry> definition in the <routine statement> in the or <procedure statement> whose execution created the current block activation.

These contexts supply the dimensionality and structuring of the resultant aggregate type. If a <parameter descriptor> or a <returns descriptor> specifies asterisk array-extents, the resultant aggregate has an array-extent of one in each dimension; otherwise, the constant array-extents of the <parameter descriptor> or <returns descriptor> supply the array-extents of the result.

Example:

```
declare SUBR entry( (*,*) fixed(15, 0) );
```

key:	meaning:
"always"	There are no error cases. These conversions satisfy the ordering referred to above.
"some"	The source must have a value in the range allowed by the resultant data type, otherwise an error situation occurs which can also raise a condition such as SIZE, OVERFLOW, or CONVERSION.

Result Source		BIT	CARDINAL	FIXED	FLOAT	CHARACTER
Type	Type:	---	-----	-----	-----	-----
BIT		-	always	some	some	some
CARDINAL		always	-	some	some	some
FIXED		always	always	-	some	some
FLOAT		always	always	always	-	always
CHARACTER		always	always	always	always	-

The conversion builtin functions and are more completely specified in section 4.7.

4.5.2 Cardinal to Fixed

Any value of type <cardinal> may be assigned to a target of type <fixed> with no loss of precision. The precision of the latter must be as large as that of the <cardinal> expression value, if not the SIZE condition is signalled.

4.5.3 Fixed to Cardinal

The conversion of values from <fixed> to <cardinal> is performed as follows. If the value is negative or non-integral, the SIZE condition is signalled. Otherwise, for non-negative integer values, the value of the source <fixed> expression is assigned to the <target>.

The precision of the <target> must be as large as that of the <fixed> expression value in order to retain all significance. If the precision of the <target> is not adequate to hold the source expression value, the SIZE condition is signalled.

The comparison of bit-string values is performed by (effectively) extending the shorter operand to the length of the longer operand (padding the shorter on the right with zero bits). The two operands are then compared bit-by-bit left-to-right with a zero bit comparing less than a one bit.

Entry values compare equal only when they correspond to the same entry and the same block activation record (see section 5.3).

File values compare equal only if they represent the same declared file (constant).

4.4.4.3 Relational Operator Results

Relational operators compare the values of their operands and yield a bit-string of length 1. The value of the result is B'1' if the relationship holds; otherwise, the value is B'0'.

4.5 Conversions

The conversions defined for infix and prefix expressions are those between the data types <fixed> and <cardinal>; these are implicit conversions. All other conversions are caused by the explicit use of conversion builtin functions.

4.5.1 Conversion Order

There is an implicit low to high hierarchy for conversions in the order

BIT, CARDINAL, FIXED, FLOAT, and CHARACTER

in the sense that a one-to-one mapping of values is defined only in the ascending direction.

There are no conversions allowed for data types entry and file. The other data types can be converted as summarized below:

with the value of the second operand.

4.4.4 The Relational Operators

The relational operators consist of the complete set:

=	equal (equivalence)
~=	not equal
<	less than
~<	not less than
<=	less than or equal
>	greater than
~>	not greater than
>=	greater than or equal

4.4.4.1 Operands for Relational Operators

Comparison is performed between values of the same data type. If the operands are of different types, they must be converted as follows:

- If either operand is arithmetic, the operand which is not arithmetic must be converted by explicit use of the `FIXED`, `CARDINAL`, or `FLOAT` built-in function.
- If one operand is a bit-string and the other is a character-string, convert the former (using the `CHARACTER` built-in function) to a character-string whose length is that of the bit-string.

All conversions are performed according to the rules given in section 4.5.

4.4.4.2 Types of Comparison

The non-equality operators are defined for all data types.

Character-string, bit-string, and arithmetic values may be compared using any of the relational operators. Since there is no ordering defined, entry and file values can only be compared using the equal operators.

Arithmetic values are compared algebraically.

The comparison of character-string values is performed by (effectively) extending the shorter operand to the length of the longer operand (padding the shorter on the right with blank characters). The two strings are then compared character-by-character left-to-right using the ASCII collating sequence.

4.4.2.2 Bit-string Infix-operator Results

For bit-string infix operators the result length is the maximum of the lengths of the two operands. If the operands are of different lengths, the shorter operand is effectively padded on the right with zero bits to the length of the longer operand.

The operations are performed on a bit-by-bit basis. Each bit of the result is developed by performing the indicated logical operation on the corresponding bits of the two operands. The following defines the logical operations for a given bit pair.

x	y	x And y	x Or y	Not x
0	0	0	0	1
0	1	0	1	1
1	0	0	1	0
1	1	1	1	0

4.4.3 The String Operator

The string operator is concatenation and is denoted by `||`. It is an infix operator that yields a string value of either type bit or character.

If both operands are bit-strings, no conversion is needed and the result is a bit-string; otherwise, the result is a character-string and both operands must be (explicitly) converted to character-strings according to the rules given in section 4.5. The lengths of the converted operands are determined by one of the following rules:

- An arithmetic operand must be converted (using the `CHARACTER` or `VARCHAR` built-in function) to character-string with length chosen according to the conversion rules for that arithmetic type.
- A bit-string operand must be converted (using the `CHARACTER` or `VARCHAR` built-in function) to a character-string whose length is at least that of the bit-string.

The result is a string whose type is the common string type of the operands, and whose length is the sum of the lengths of the operands. If either operand has the `<varying>` attribute, the result will also have it.

The value of the result is the value of the first operand concatenated

conditions	result
$x=0$ and $y>0$	0
$x=0$ and $y\leq 0$	signal error condition
$x\neq 0$ and $y=0$	1
$x<0$ and y is not a <constant integer>	signal the error condition

For other cases the result is an approximation to the actual value.

4.4.2 Bit-string Operators

The bit-string operators are:

Symbol	Operator	Form
~	complement	prefix
	inclusive or	infix
&	and	infix

Bit-string operators require bit-string operands; they do not force conversion of their operands to bit-strings, however explicit conversions can be written using builtin functions. The latter operate according to the rules given in section 4.5. The lengths of the converted operands are defined by the following rules:

- A character-string operand must be converted (using the BIT builtin function) to type bit-string, normally the same length as the character-string.
- An arithmetic operand must be converted (using the BIT built-in function) to type bit-string with length as defined in section 4.7.

4.4.2.1 Bit-string Prefix-operator Results

The complement operator is the only prefix operator.

The result of the complement operator has the same bit-string attributes as the operand. The result value is the complement of the value of the converted operand (each 1 bit becomes a 0 bit, and each 0 bit becomes a 1 bit).

allowed, and that as many fractional digits are preserved as are allowed by the implementation. Use of these values as operands of other fixed-point computations can lead to states of computation that produce the fixedoverflow condition.

Example:

```

Declare (A, B) fixed (7, 0),
        C    fixed (7, 0);
...  A/B + C  ...

```

The precision of the quotient is (63, 56); for many values of A, B, and C the fixedoverflow condition could occur. It does so because the division yields a fraction; then the addition attempts to align the radix points by scaling the value of C. As a result of the latter, the precision of the scaled quantity exceeds the limit (N=63) of the machine-system.

The DIVIDE built-in function described in section 4.7 can be used to control the precision of the result of fixed-point (or floating-point) division.

Note: if the relation $p+q-s = N$ holds, the quotient is an integer. For example, if X and Y are integers of maximum precision (N, 0), then the quotient of X / Y has precision (N, 0).

Implementation note: A max_precision option could be a desirable procedure <entry option>.

Power operator (**).

If the second operand is a constant or a <decimal integer> whose value is denoted by E, the result is fixed-point only if $((p+1)*E-1) \leq N$; otherwise, the first operand is converted to floating-point and the rules for algebraic exponentiation apply.

In cases for which the result is fixed-point, the precision is given by:

$$((p+1) * E - 1, q * E)$$

The value of the result is the value of the first operand multiplied by itself E-1 times.

In the following cases, if X is the first operand and Y is the second operand, $X ** Y$ is defined as follows:

Format:

LOG2(<expression>)

The value returned has the same data type as the operand. If <expression> is an array or structure, then the logarithm of each member is returned.

4.7.3.3 SIN

The SIN built-in function will find the sine of a given value, which is expressed in radians.

Format:

SIN(<expression>)

The <expression> term must be either fixed-point or floating-point. The value returned is the sine of <expression>, where <expression> is in radians. If <expression> is an array or structure, then the sine of each member is returned.

4.7.3.4 SQRT

The SQRT built-in function will find the square root of a given value.

Format:

SQRT(<expression>)

The <expression> term must be either fixed-point or floating-point, and positive. The value returned is the square root of <expression>. If <expression> is an array or structure, then the square root of each member is returned. The ERROR condition will be raised if any member of <expression> is negative.

4.7.3.5 TAN

The TAN built-in function will find the tangent of a given value, which is expressed in radians.

Format:

TAN(<expression>)

The <expression> term must be either fixed-point or floating-point. The value returned is the tangent of <expression>, where <expression> is in radians. If <expression> is an array or structure, then the tangent of each member is returned. The ERROR condition will be raised if any member of <expression> represents a value of either 90 degrees or 270 degrees.

4.7.2.19 TRUNC

The TRUNC built-in function will truncate a given value to an integer.

Format:

TRUNC(<expression>)

The result of TRUNC is not larger than <expression> in absolute value. If <expression> is an array or structure, then the TRUNC of each member is returned.

The function first determines if a value is positive, negative, or equal to 0. If the value is negative, TRUNC returns the smallest integer that is not less than that value. If the value is positive or zero, TRUNC returns the largest integer that does not exceed that value.

In tabular form:

<u>magnitude of <expression></u>	<u>value produced</u>
less than zero	CEIL(<expression>)
zero or greater	FLOOR(<expression>)

4.7.3 Mathematical Built-in Functions

4.7.3.1 COS

The COS built-in function will find the cosine of a given value, which is expressed in radians.

Format:

COS(<expression>)

The <expression> term must be either fixed-point or floating-point. The value returned is the cosine of <expression>, where <expression> is in radians. If <expression> is an array or structure, then the cosine of each member is returned.

4.7.3.2 LOG2

The LOG2 built-in function will find the binary (base 2) logarithm of a given floating-point or fixed-point value.

Format:

```
MULTIPLY( <expression>, <expression>, <result precision>
          [, <result scale> ] )
```

The first two <expression>s are multiplied together to produce a result with precision specified by <result precision>, and with scale specified by <result scale>. If the type of the result is floating-point, the <result scale> term is not allowed. If the first two <expression>s are both arrays or structures, then the product of each member is returned. The rules governing conformance and promotion (4.6) apply when multiplying the two <expression>s together.

4.7.2.17 PRECISION

The PRECISION built-in function will convert an arithmetic value to a specified precision.

Format:

```
PRECISION( <expression>, <result precision>
           [, <result scale> ] )
```

The <expression> term must be either a fixed-point or a floating-point value. The <result precision> term specifies the precision and the <result scale> term the scale. If the type of the value of <expression> is floating point, only the <result precision> term is allowed. If <expression> is an array or structure, then the PRECISION function is applied to each member.

4.7.2.18 SUBTRACT

The SUBTRACT built-in function will control the precision of a subtraction operation.

Format:

```
SUBTRACT( <expression>, <expression>, <result precision>
          [, <result scale> ] )
```

The second <expression> is subtracted from the first <expression> and the resulting value will have precision as specified by the <result precision> term and scale as specified by the <result scale> term. If the resulting value is of type floating-point, then only the <result precision> term may be specified. If <expression> is an array or structure, then the SUBTRACT of each member is returned. The rules governing conformance and promotion (4.6) apply when subtracting the two <expression>s.

returned is B'0'. If <expression> is an array, then a bit-string array (of length one) of values is returned.

4.7.2.13 MAX

The MAX built-in function will extract the higher-valued expression from a given pair expressions. If the two <expression>s are arrays, then an array consisting of the higher value of each pair is returned.

Format:

MAX(<expression>, <expression>)

4.7.2.14 MIN

The MIN built-in function will extract the lower-valued expression from a given pair of expressions, possibly arrays of matching extents. If the two <expression>s are arrays, then an array consisting of the smaller value of each pair is returned.

Format:

MIN(<expression>, <expression>)

4.7.2.15 MOD

The MOD built-in function will determine the remainder resulting from dividing one arithmetic quantity by another.

Format:

MOD(<expression>, <expression>)

The value returned is zero if the second <expression> is zero. Otherwise the value is given algebraically by

$$\text{MOD}(x, y) = x - y * \text{FLOOR}(x / y)$$

If <expression> is an array or structure, then the MOD of each member is returned. The rules governing conformance and promotion (4.6) apply when dividing the first <expression> by the second.

4.7.2.16 MULTIPLY

The MULTIPLY built-in function controls the precision of the result of a multiplication operation.

Format:

```
MULTIPLY( <expression>, <expression>, <result precision>
          [, <result scale> ] )
```

The first two <expression>s are multiplied together to produce a result with precision specified by <result precision>, and with scale specified by <result scale>. If the type of the result is floating-point, the <result scale> term is not allowed. If the first two <expression>s are both arrays or structures, then the product of each member is returned. The rules governing conformance and promotion (4.6) apply when multiplying the two <expression>s together.

4.7.2.17 PRECISION

The PRECISION built-in function will convert an arithmetic value to a specified precision.

Format:

```
PRECISION( <expression>, <result precision>
           [, <result scale> ] )
```

The <expression> term must be either a fixed-point or a floating-point value. The <result precision> term specifies the precision and the <result scale> term the scale. If the type of the value of <expression> is floating point, only the <result precision> term is allowed. If <expression> is an array or structure, then the PRECISION function is applied to each member.

4.7.2.18 SUBTRACT

The SUBTRACT built-in function will control the precision of a subtraction operation.

Format:

```
SUBTRACT( <expression>, <expression>, <result precision>
          [, <result scale> ] )
```

The second <expression> is subtracted from the first <expression> and the resulting value will have precision as specified by the <result precision> term and scale as specified by the <result scale> term. If the resulting value is of type floating-point, then only the <result precision> term may be specified. If <expression> is an array or structure, then the SUBTRACT of each member is returned. The rules governing conformance and promotion (4.6) apply when subtracting the two <expression>s.

Format:

INDEX_FIRST(<expression>)

This function will examine <expression>, a bit-string array, and return the value of the index of the first item or element of <expression> containing any bit with a value of one. If all items of <expression> are equal to B'0', a value of zero will be returned.

4.7.2.10 INDEX_MAX

The INDEX_MAX built-in function will determine the index of the largest value in a single dimensioned array.

Format:

INDEX_MAX(<expression>)

This function will examine <expression>, a one dimensioned array, and return the value of the index of the array element member whose value is the largest. In the event of more than one array member having the same largest value, the smallest index value will be returned.

4.7.2.11 INDEX_MIN

The INDEX_MIN built-in function will determine the index of the smallest value in a single dimensioned array.

Format:

INDEX_MIN(<expression>)

This function will examine <expression>, a one dimensioned array, and return the value of the index of the array element member whose value is the smallest. In the event of more than one array member having the same smallest value, the smallest index value will be returned.

4.7.2.12 INTERVAL_TEST

The INTERVAL_TEST built-in function will perform an interval comparison test.

Format:

INTERVAL_TEST(<expression>, <lower limit>, <upper limit>)

This function returns a value of type bit-string (length one). If the value of <expression> is greater than or equal to <lower limit>, and less than or equal to <upper limit>, then the value returned is B'1', otherwise the value

If the <result precision> term is less than the precision of the source <expression>, the OVERFLOW condition will be raised. Otherwise the FIXED source <expression> is converted to the equivalent floating-point representation and assigned to the target.

If the source is CHARACTER:

The precision of the converted result will be specified by the <result precision> term, unless omitted. If omitted, the precision of the converted result is derived as follows: Let len be the declared length of the source <expression>, then the precision of the converted result is equal to $\text{CEIL}(3.32 * \text{len})$.

The source <expression> must consist only of characters as specified by the definition of a <floating-point constant>. It is then converted to its base two radix equivalent. Thus the converted result of a given character string is simply its base two radix equivalent with precision as specified above.

If the <result precision> term is specified, and it is less than $\text{CEIL}(3.32 * \text{len})$, then the SIZE condition is raised and the converted result is undefined. If the source <expression> consists of characters other than allowed for a <floating-point constant>, the CONVERSION condition will be raised. The CONVERSION condition will also be raised if the <result precision> term is omitted and the length of the source <expression> is null.

4.7.2.8 FLOOR

The FLOOR built-in function will determine the largest integer that does not exceed a given value.

Format:

`FLOOR(<expression>)`

The <expression> term must be of type arithmetic, and the value returned will be the largest fixed-point scaled zero integer that does not exceed <expression>. If <expression> is an array or structure, then the FLOOR of each member is returned.

4.7.2.9 INDEX_FIRST

The INDEX_FIRST built-in function will determine the smallest index position of a one bit in a bit-string array.

If the source is CHARACTER:

The precision of the converted result will be specified by the `<result precision>` term, unless omitted. If omitted, the precision of the converted result is derived as follows: Let `len` be the declared length of the source `<expression>`, then the precision of the converted result is equal to $\text{CEIL}(3.32 * \text{len})$.

The source `<expression>` must consist only of characters as specified by the definition of a `<decimal fixed-point constant>`. It is then converted to its base two radix equivalent. Thus the converted result of a given character string is simply its base two radix equivalent with precision as specified above.

If the `<result precision>` term is specified, and it is less than $\text{CEIL}(3.32 * \text{len})$, then the SIZE condition is raised and the converted result is undefined. If the source `<expression>` consists of characters other than allowed for a `<decimal fixed-point constant>`, the CONVERSION condition will be raised. The CONVERSION condition will also be raised if the `<result precision>` term is omitted and the length of the source `<expression>` is null.

4.7.2.7 FLOAT

The FLOAT built-in function controls the precision of the result of a conversion to floating-point.

Format:

```

FLOAT( <expression> [, <result precision> ] )

```

The `<expression>` is the value to be converted to floating-point. The `<result precision>` term, when present, specifies the precision of the result. If `<expression>` is an array or structure, then the conversion of each member is returned.

The rules for conversion to FLOAT depend on the type of source data. The rules follow:

If the source is BIT:

The source `<expression>` must first be converted to CARDINAL.

If the source is CARDINAL:

The source `<expression>` must first be converted to FIXED.

If the source is FIXED:

<result scale> term, when present, specifies the scale of the result. If the result is a floating point value, then only the <result precision> term may be specified. If <expression> is an array or structure, then the division of each member is returned. The rules governing conformance and promotion (4.6) apply when dividing the first <expression> by the second.

4.7.2.6 FIXED

The FIXED built-in function controls the precision of the result of a conversion to fixed-point.

Format:

```
FIXED( <expression> [, <result precision> [, <result scale> ] ] )
```

The <expression> is the value to be converted to fixed-point. The <result precision> term specifies the precision, and the <result scale> term the scale of the result. If <expression> is an array or structure, then the conversion of each member is returned.

The rules for conversion to FIXED depend on the type of the source data. The rules follow:

If the source is BIT:

(1) The source <expression> must first be converted to CARDINAL.

If the source is CARDINAL:

The precision of the result will be specified by the <result precision> term, unless omitted. If omitted, the precision of the converted result is that of the source <expression>. See section 4.5.1 for further discussion.

If the source is FIXED:

In this case only the precision is altered. See the PRECISION built-in function for further discussion.

If the source is FLOAT:

If the optional <result scale> term is omitted, a fixed-point scaled zero value is returned. If the <result precision> term is omitted, the precision of the source <expression> will be the precision of the result. If the <result precision> term is specified, and the value of the floating-point <expression> cannot be contained in the range of values allowed by the <result precision> and <result scale> of the target, the SIZE condition is raised. Otherwise, the source <expression> is converted to a fixed-point value with precision and scale as specified.

The source <expression> must consist only of blanks and/or decimal digits. It is then converted to its base two radix equivalent. Thus the converted result of a given character string is simply its base two radix equivalent with precision as specified above.

If the <result precision> term is specified, and it is less than $\text{CEIL}(3.32 * \text{len})$, then the SIZE condition is raised and the converted result is undefined. If the source <expression> consists of characters other than blanks and/or decimal digits, the CONVERSION condition will be raised. The CONVERSION condition will also be raised if the <result precision> is unspecified and the length of the source <expression> is null.

If the source is FIXED:

The precision of the result will be specified by the <result precision> term, unless omitted. If omitted, the precision of the converted result is that of the source <expression>. See section 4.5.2 for further discussion.

If the source is FLOAT:

The source <expression> must first be converted to FIXED.

4.7.2.4 CEIL

The CEIL built-in function determines the smallest integer that is greater than or equal to a given fixed or floating point expression.

Format:

`CEIL(<expression>)`

The value returned by CEIL will be the smallest fixed-point scaled zero integer number that is greater than or equal to the fixed or floating point value represented by <expression>. If <expression> is an array or structure, then the CEIL of each member is returned.

4.7.2.5 DIVIDE

The DIVIDE built-in function controls the precision of the result of a divide operation.

Format:

`DIVIDE(<expression>, <expression>, <result precision>
[, <result scale>])`

The first <expression> is the dividend and the second <expression> is the divisor. Both must be arithmetic and of the same type. The <result precision> term specifies the precision of the result, while the

AD-A040 468

GOODYEAR AEROSPACE CORP AKRON OHIO
SPECIFICATION FOR A STARAN PROGRAMMING LANGUAGE.(U)
APR 77 R G LANGE, H CHEESEMAN, E W DAVIS
GER-16347

F/G 9/2

F30602-76-C-0200

UNCLASSIFIED

RADC-TR-77-131

NL

2 OF 2
AD
A040468



result is floating point, only the <result precision> term may be specified. If <expression> is an array or structure, then the addition of each member is returned. The rules governing conformance and promotion (4.6) apply when adding the two <expression>s together.

4.7.2.3 CARDINAL

The CARDINAL built-in function will control the precision of the result of a conversion to cardinal.

Format:

CARDINAL(<expression> [, <result precision>])

The <expression> is the value to be converted to cardinal. The <result precision> term specifies the precision of the result. If <expression> is an array or structure, then the conversion of each member is returned.

The rules for conversion to CARDINAL depend on the type of the source data. The rules follow:

If the source is BIT:

(1) If the length of the source <expression> is equal to or less than the <result precision> value, or the latter is omitted, the result value is the value of the source <expression>.

(2) If the length of the source <expression> is greater than the specified <result precision> value, the SIZE condition is raised and the converted result is undefined.

If the source is CARDINAL:

(1) If the precision of the source <expression> is equal to or less than the <result precision> value, or the latter is omitted, the result value is the value of the source <expression>.

(2) If the precision of the source <expression> is greater than the specified <result precision> value, the SIZE condition is raised and the converted result is undefined.

If the source is CHARACTER:

The precision of the converted result will be specified by the <result precision> term, unless omitted. If omitted, the precision of the converted result is derived as follows: Let len be the declared length of the source <expression>, then the precision of the converted result is equal to $\text{CEIL}(3.32 \cdot \text{len})$.

Example:

```
DOIT: procedure(K, X) returns(fixed(15, 0) );
```

In this example, DOIT is an entry to the <procedure> headed by the <procedure statement>. Invocation of DOIT results in a fixed-point value. The entry requires two arguments which are associated with the parameters K and X.

5.3.2 The Routine Statement

Syntax:

```
<routine statement> ::= <label prefix> ROUTINE
    [ ( [ <parameter list> ] ) ]
    [ <entry option> ] ;
```

The ROUTINE statement provides an alternate entry to a <procedure> and allows sharing of the <block> scope for variables declared in the procedure. The statements following a <routine statement> to the next <routine statement> or the <endproc statement> that ends the procedure are the body for execution of the entry.

Upon invocation of the entry associated with the ROUTINE statement, a new block activation of the containing <procedure> occurs and the arguments of the <function reference> or <call statement> are paired with the corresponding parameter in the <parameter list>.

If control reaches a <routine statement> as a result of completing the execution of the preceding statement, control is returned from the containing <procedure> without a <function value> as though a <return statement> was encountered.

An "entry constant" is declared by the <label prefix> of a <routine statement> (rather than declaration of a label constant), thus a <goto statement> cannot transfer control to a <routine statement>.

Example:

```
FCN: routine(X,Y,Z) returns(cardinal(8));
```

In this example, FCN is an entry whose invocation results in an integer value. The entry requires three arguments which are associated with the parameters X, Y, and Z.

The number of <identifier>s in the <parameter list> must equal the number of <expression>s in any <argument list> of any <call statement> or <function reference> that invokes this entry.

5.3.1 Procedure Statement

Syntax:

```

<procedure statement> ::= <label prefix> PROCEDURE
    [ ( [ <parameter list> ] ) ]
    [ <procedure option> ]... ;

<procedure option> ::= <entry option> | RECURSIVE

<entry option> ::= <reducible option>
    | <returns attribute>

<parameter list> ::= <identifier>
    [ , <identifier> ] ...

<reducible option> ::= <irreducible attribute>
    | <reducible attribute>
<irreducible attribute> ::= IRREDUCIBLE
<reducible attribute>   ::= REDUCIBLE
<returns attribute>    ::= RETURNS
    ( <returns descriptor> )
<returns descriptor>   ::= <descriptor>
    [ , <descriptor> ]...
<descriptor> ::= <level> [ <attribute set> ]
    | [ <level> ] <attribute set>

```

A procedure is constructed with a heading, body, and ending. A <procedure statement> is the heading for a <procedure> and denotes an entry to the <procedure>. When control is transferred to the entry a new block activation of the <procedure> occurs, and the arguments of the <function reference> or <call statement> are associated with the (formal) parameters in the <parameter list>. The specification of <procedure> activation is in section 5.6 and the specification of argument and parameter semantics is in section 4.2.

A <procedure statement> is not executed after its lexically preceeding statement; the successor to the latter is the statement logically following the corresponding <endproc statement>.

An "entry constant" is declared by the <label prefix> of a <procedure statement> (rather than declaration of a label constant), thus a <goto statement> cannot transfer control to a <procedure statement>.

5.3 Procedures

Syntax:

```

<procedure> ::= <procedure statement>
               <procedure body> <endproc statement>

<procedure body> ::= <procedure component>
                    [ <routine statement>
                      <procedure component> ]...

<procedure component> ::= <statement> ...

<endproc statement> ::= ENDPROC <closure identifier> ;
<closure identifier> ::= <identifier>

```

A <procedure> can have multiple entries corresponding to multiple bodies, called <procedure component>s; the additional entries are defined by ROUTINE statements within the procedure. Each code body is distinct and has a single entry and single exit for flow of control. The entry is invoked by either a <call statement> or a <function reference> that references an entry value corresponding to the <label prefix> on either a <procedure statement> or an <routine statement>.

Semantics rules that apply to both <procedure statement>s and <routine statement>s are:

The <parameter list> contains <identifier>s which each identify a level-one variable declared in the <procedure> headed by the immediately containing <procedure statement>.

The <identifier> written in the <label prefix> of the <procedure statement> or <routine statement> is given the attributes <entry> and <constant>. The scope of these identifiers is that of the containing procedure, or external in the case that there is no containing procedure.

The <label prefix> cannot contain a <case prefix>.

If control passes to the <procedure statement> or <routine statement> by the execution of a <call statement>, the entry being defined cannot have a <returns attribute>.

If control passes to the <procedure statement> or <routine statement> by the evaluation of a <function reference>, the entry being defined must have a <returns attribute>.

5. THE DYNAMIC BEHAVIOR OF A PROGRAM

5.1 The Flow of Control

A program is executed by a processor that follows a single path, known as the flow of control, through the statements of the program. The program determines the flow of control by the use of <call statements>s, <conditional statement>s, <group>s, <function reference>s, and <goto statement>s.

5.2 A Program

A program is a set of <program unit>s together with their operating environment including the data sets referenced by the program. The effect of the program is the set of all changes made by the program to any of the data sets.

The set of <program unit>s that are part of the program is determined dynamically by the execution of the program, and by linking facilities which resolve references to items having the "external" scope attribute.

The flow of control between <program unit>s is determined by the execution of CALL statements and function references. These reference the EXTERNAL entries defined by <program unit>s.

Format:

ALLOCATION(<identifier>)

The <identifier> must be an unsubscripted array name, a major structure (level-1) name or an element variable name. It must have the controlled attribute. If storage has not been allocated to <identifier>, the returned value is zero. If storage is allocated for the <identifier>, the value is one. The returned value is of type CARDINAL(8).

4.7.4 Array Built-in Functions

4.7.4.1 DIM

The DIM built-in function will determine the current size of a specific bound of a given array.

Format:

DIM(<reference>, <expression>)

The <reference> is an array and <expression> references a particular dimension of the array. DIM returns the current value of the specified dimension of an array.

4.7.4.2 SELECT

The SELECT built-in function provides the ability to select specific elements of an array for processing.

Format:

SELECT(<reference>, <reference>)

The first <reference> represents any array, and the second <reference> is a bit-string array of length one with the same dimension as the first array <reference>. Only those items in the first <reference> corresponding to bit-string values of B'1' in the second <reference> are selected for processing.

4.7.4.3 SUM

The SUM built-in function will find the sum of all the elements of a given array.

Format:

SUM(<expression>)

The <expression> is an array whose elements are added together and returned by the SUM function.

4.7.5 Miscellaneous Built-in Functions

Allocation

The ALLOCATION built-in function will determine whether or not storage is allocated for a given variable having the attribute controlled.

6.1.2 Active and Inactive DO-Loops

A DO-loop is either active or inactive. Initially inactive, a DO-loop becomes active only when its DO statement is executed.

Once active, the DO-loop becomes inactive only when:

- (1) its termination condition is satisfied
- (2) a RETURN statement is executed within the range of the loop
- (3) An EXIT statement is executed within the range of the loop that identifies this loop to be terminated.

When a DO-loop becomes inactive, the <index> of the DO-loop retains its last defined value.

6.1.3 Executing an Iterative DO Statement

The effect of executing a DO statement is to perform the following two steps (not necessarily in sequence):

(1) an initial parameter called m1 is set using the current value of the first <expression> in <step definition>; a terminal parameter called m2 is set using the current value of <limit> in <iteration control>; and an incrementation parameter called m3 is set from the current value of <increment> in <iteration control>. If the <increment> does not appear in <iteration control>, m3 has a value of one.

(2) The <index> becomes defined with the value of the initial parameter m1.

At the completion of execution of the DO statement, loop control processing begins.

6.1.4 Loop Control Processing

Loop control processing determines if execution of the range of the DO-loop <group body> is required. If <index> > m2 and m3 >= 0, or if <index> < m2 and m3 <= 0, then the DO-loop becomes inactive, and execution continues with the execution of the next executable statement following the end of the DO-loop. Otherwise, the first statement in the range of the DO-loop is executed.

6. CONTROL STATEMENTS

Flow of control statements are those associated with altering the normal sequential execution of program statements. For example, flow of control can be changed by loops, conditional statements, procedure calls, and the GOTO statement.

6.1 Iterative DO-Loop

Both forms of the DO are used to specify a loop. The <iterative loop> is called a DO-loop. The <repetitive loop> is called a DO WHILE-loop and will be fully described in section 6.2.

Syntax:

```

<iterative loop> ::= DO <step definition> <group body>
                      ENDDO [ <identifier> ] ;
<step definition> ::= <index> = <expression>
                      <iteration control> ;
<index> ::= <identifier>
<iteration control> ::= TO <limit> [ BY <increment> ]
<limit> ::= <expression>
<increment> ::= <expression>
<group body> ::= <basic statement>...

```

Each form of the DO requires a corresponding ENDDO to denote the range of the loop.

The evaluation of all occurrences of <expression> must be fixed-point integer scalar values. The <index> must be a scalar variable of type <fixed> or <cardinal> and must yield an existing generation of storage.

6.1.1 Range of a DO-Loop

The range of a DO-loop consists of the executable statements from and including the first executable <basic statement> following the DO statement that specifies the DO-loop, to and including the corresponding ENDDO statement for the DO-loop. Transfer of control into the range of a DO-loop is not permitted.

with the execution of the statement following the <procedure>.

5.6.2 On Units

The execution of an <on statement> causes the <on unit> to be established, but does not cause execution of the <on unit> (see section 9.2). The establishment of an <on unit> is the process of associating it with the block activation that contains the <on statement>. The environment of an <on unit> activation is the block activation that established the <on unit>.

A condition can be signalled by either the execution of a <signal statement> or by detection of the condition during some aspect of program execution. Examples of the latter are <reference> evaluation or <expression> evaluation. Control enters an established <on unit> when the condition identified by the condition name is signalled. An <on unit> consisting of a single statement behaves as if it were a <block>, and execution of an <on unit> is effectively a block activation.

Control returns from an <on unit> by the execution of a nonlocal goto or when control reaches the end of the <on unit>.

Further discussion of conditions is given in section 9.

9 for a discussion of <on unit>s and the flow of control.

5.5 The Flow of Control Within an Activation

The control within a block activation proceeds from <statement> to <statement> in the same order in which the statements appear in the text of the <block>, except as influenced by actions of the <statement>s.

The components of a statement are evaluated in an order defined in sections 5 through 9 where the syntax and semantics of each type of statement are defined. The order of evaluation of <expression>s is given in section 4.1. The control within a <group> is specified by the options of the DO statement which begins the <group> and by <statement>s within the <group>.

A <goto statement> may not transfer control between two <procedure component>s of a multiple-entry <procedure> or between two <block>s. It may transfer control to a labeled <basic statement> within the <block> by referencing a name declared by a <label prefix> appearing on a statement other than a <routine statement> or <procedure statement> within the <block>. This form of <goto statement> is called a local goto.

5.6 Inter-block Flow of Control

A <goto statement> written to transfer control to a <statement> in a dynamically preceding and (statically) containing block activation is called a non-local goto. The non-local goto is allowed only as an exit from an <on unit> to terminate the latter.

5.6.1 Procedures

Control can only pass to a <procedure> when one of its entries is invoked by a <function reference> or <call statement>. Control returns from the <procedure> to the invoking <block> by the execution of a <return statement>, a reaching <routine statement>, or the execution of the <endproc statement> which terminates the <block>.

A <procedure> invoked as a function by a <function reference> must return control by the execution of a <return statement> containing a <function value>. A <procedure> invoked as a subroutine by a <call statement> cannot return control by the execution of a <return statement> that contains a <function value>.

If control reaches a <procedure statement>, except as a result of an invocation of the <procedure>, it passes around the <procedure> and continues

Some optimization information for the compiler is provided by the use of the <varying>, <memory>, <aligned>, and <unaligned> attributes in a <returns descriptor>. The semantics of the language are not affected by such information.

In a return statement, the <function value> must have an attribute set identical to that of the <returns descriptor> of the <returns attribute> written on the <procedure statement> or <routine statement> for each invocation of the entry values represented by the item.

5.4 Procedures and Dynamic Block Structures

5.4.1 Procedure Activation

Whenever control is transferred to a procedure <block>, the <block> becomes activated. The activation of the procedure <block> remains in effect until control returns from the <block>. Note that at least one <block> is always active, i.e., the first one that control entered (the initial <program unit>). It is possible for several <block>s to be active at the same time, since <block>s may be nested and <procedure>s may call each other. However, control is only in one <block> at any given snapshot of time.

The dynamic relationship between <block>s is a function of the order in which the <block>s were activated. Thus <block> X is the dynamic predecessor of <block> Y, and Y is the dynamic descendent of X, if control passes from <block> X by invocation of <block> Y.

5.4.2 The Environment of an Activation

Every block activation has a static environment, it is the <block>'s immediate statically containing <block>. Since <program unit>s and <external procedure>s have no containing <block>, their static environment is null.

If statements in a block reference automatic variables, parameters, or entry constants declared in a containing <block>, control must know which of several possible activations of the containing <block> it is to reference. The dynamic order of <block> activations provides this information.

The environment of an activation of an <on unit> is the block activation that established the <on unit> via an <on statement>. See sections 5.6.2 and

<descriptor> members have no names.

Example:

```
declare X entry(1, 2 fixed(31, 0), 2 bit(256),
               (8, 8) bit(8) );
```

The entry X has 2 parameters. The first is a structure containing an integer and a bit string. The second parameter is an eight-by-eight array of bit strings.

5.3.3.1 Irreducible Entry Attribute

An irreducible entry is designated by an entry value represented by a name declared with the <irreducible attribute> as described in section 4.2.9. Entry values represented by a name declared with the <irreducible attribute> are not restricted. The <irreducible attribute>'s only significance is to force the entry values to be invoked once for each evaluation of an <entry reference> in either a <call statement> or <function reference>. See section 4.1.3 on order and frequency of evaluation.

5.3.3.2 Reducible Entry Attribute

A reducible entry is designated by an entry value represented by a name declared with the <reducible attribute> as described in section 4.2.9. Entry values represented by a name declared with the <reducible attribute> are not restricted. The <reducible attribute>'s only significance is to possibly reduce the number of invocations of the entry values.

5.3.3.3 Returns Attribute

Entry values may be represented by a name declared with a <returns attribute>. However, such a name cannot be invoked by the execution of a <call statement>. Entry values represented by a name declared with the <returns attribute> are not restricted. The <returns attribute>'s only significance is to limit the invocation of such entry values to the evaluation of <function reference>s.

When used in a <returns descriptor>, <extent>s must be either <cvi expression>s (constant value integer expressions) or asterisks. The <extent>s of all values returned by a function are the same, if the <extent>s are <cvi expression>s. However, the <extent>s may differ from one function invocation to the next; if the <extent>s are in the form of asterisks. Note that if any <extent> expressions are in the form of asterisks, then all such expressions must be in the form of asterisks.

5.3.3 Entry Attributes

Syntax:

```

<entry> ::= ENTRY [ (
    [ <parameter descriptor list> ] ) ]
    [ <entry option> ]

<parameter descriptor list> ::= <parameter descriptor>
    [ , <parameter descriptor> ]...

<parameter descriptor> ::= <descriptor>

```

A data item declared with the <entry> attribute may only have entry values.

An entry value without a <parameter descriptor list> item will be invoked for the following cases:

- (1) The entry value is identified by the <entry reference> of a <call statement>.
- (2) The entry value is identified by the <entry reference> of a <function reference> with a null <argument list> (the form "ENTRY()").

The form "ENTRY()" and the form "ENTRY" are equivalent, except that "ENTRY()" represents a complete <attribute>, whereas "ENTRY" is an incomplete <attribute>.

The <parameter descriptor list> is significant only when an entry value represented by the item is invoked; it does not restrict the values that may be represented by the <entry> item.

For each of the entry values represented by this item and for each invocation of those entries, the <parameter descriptor list> must produce a list of declarations that is equivalent to the actual declaration of the parameters for the entry invoked. To be equivalent, the declarations must have exactly the same <attribute set>s.

The syntax of a <descriptor> of a structure and that of a <declaration component> of a structure variable are identical, except that a <descriptor> has no name. Similarly the members of a <descriptor> and the members of a <declaration component> are declared identically, except that

If there are both true and false values in the bit-string array result of the <expression> evaluation, then selective processing may be used during execution of the <basic statement>s in the <true part> and/or <else part> portions of the <ifarray statement>.

6.4.2.3 IFARRAY Statement Selective Processing

The bit-string array result of the <expression> evaluation may be utilized to select elements of an array if the array selection symbol "&" is used. If "&" is used as selection subscript in <basic statement>s in the <true part>, then only those array elements corresponding to true (any 1 bits) values of the bit-string array result are selected for execution. If "&" is used in <basic statement>s in the <else part>, then only those array elements corresponding to false (all 0 bits) values of the bit-string array result are selected for execution. In either case, the statements in both the <true part> and <else part> in the body of the <ifarray statement> are executed.

Three examples illustrate the above:

Let A, B, C, and D be one dimensional arrays of equal length and type. Let E be a bit-string array of the same length as the arrays A, B, C, and D.

```
IFARRAY e THEN
    a[&] = b + c;
    b = a;
ELSE
    a[&] = c + d;
    d = a;
ENDIF;
```

Example 1.

Let bit-string array E contain both true and false values.

After execution of the above statements, $A[i]$ will be equal to $B[i]+C[i]$ only if $E[i]$ is true. $A[i]$ will be equal to $C[i]+D[i]$ only if $E[i]$ is false. $B[i]$ is set equal to $A[i]$ for all i , and $D[i]$ is set equal to $A[i]$ for all i .

Example 2.

Let bit-string array E contain only true values.

After execution of the above statements, $A[i]$ will be set equal to $B[i]+C[i]$ for all i , and $B[i]$ is set equal to $A[i]$ for all i . The two statements between ELSE and ENDIF are not executed.

range of each of the two possible paths of program flow.

6.4.1.1 Body of an IF Statement

The body of an <if statement> consists of the executable statements from and including the <if clause> statement, to and including the ENDIF statement.

6.4.1.2 Executing an <if clause>

The effect of executing an <if clause> is to evaluate the <expression> producing a scalar value of type bit-string. At the completion of execution of the <if clause> statement, flow of control processing begins.

6.4.1.3 IF Statement Flow of Control Processing

Once the <if clause> <expression> has been evaluated, the bit-string result is examined for any one bits. If there are any one bits, execution continues with execution of the first executable statement following the THEN keyword until either an ELSE or an ENDIF is reached. Execution will then continue with execution of the first executable statement following the ENDIF.

If there are no one bits, then execution continues with execution of the first executable statement following the ELSE keyword until an ENDIF is reached. If there is no ELSE keyword present, then execution will continue with execution of the first executable statement following the ENDIF.

Transfer of control out of a <if statement> via a <goto statement> is not permitted.

6.4.2 The IFARRAY Statement

The <ifarray statement> provides for the execution of all the instructions within the body of the IFARRAY statement.

6.4.2.1 Body of an IFARRAY Statement

The body of an <ifarray statement> consists of the executable statements from and including the <ifarray clause> through the ENDIF.

Transfer of control out of a <ifarray statement> via a <goto statement> is not permitted.

6.4.2.2 Executing an <ifarray clause>

In the <ifarray clause>, the <expression> must evaluate to an array value of type bit-string.

6.4 Conditional Statements

The <conditional statement> specifies which of two possible program paths is to be executed next, depending on the value of a given logical expression.

Syntax:

```

<conditional statement> ::= <if statement>
                        | <ifarray statement>
                        | <logical if statement>
<if statement> ::= <if clause> <if body> <endif>
<ifarray statement> ::= <ifarray clause>
                        <if body> <endif>
<if clause> ::= IF <expression>
<ifarray clause> ::= IFARRAY <expression>
<if body> ::= <true part> [ <else part> ]
<true part> ::= THEN <basic statement>...
<else part> ::= ELSE <basic statement>...
<endif> ::= ENDIF [ <identifier> ] ;
<logical if statement> ::= <logical if>
                        <single statement>
<logical if> ::= IF ( <expression> )

```

The evaluation of <expression> found in <if clause> and in <logical if> must yield a scalar value of type bit-string. The <true part> is executed if the <expression> has any one bits in the bit-string. The optional <else part> is executed if there are no one bits in the bit-string.

In the <ifarray clause>, the <expression> is an array of type bit-string; the <true part> or <else part> may be executed for each element of the array.

If the optional <identifier> is specified on the ENDIF statement, then the corresponding IF statement must have a <label prefix> identical to <identifier>.

6.4.1 The IF Statement

The <if statement> provides for the execution of statements within the

After execution of the above statements, J equals 10 if I equals 0, J equals 7 if I equals 1,3 or 4, and J equals 3 if I is not equal to 0, 1,3 or 4.

6.3.1 Body of DO CASE-group

The body of a DO CASE-group consists of the executable statements from and including the first executable <basic statement> following the DO CASE statement that specifies the DO CASE-group, to and including the ENDCASE statement that terminates the DO CASE-group.

6.3.2 Executing a DO CASE Statement

The effect of executing a DO CASE statement is to perform the following steps in sequence:

- (1) The DO CASE <expression> is evaluated and must be scalar and of type <cardinal> or <fixed> scaled zero.
- (2) The value of the DO CASE <expression> is compared with the known constant values of the <case prefix> <expression>s in the <case body> of the DO CASE-group.

If the value of the <expression> in <case heading> does not match any of the known constant values of the <case prefix> <expression>s, control is transferred to the ELSECASE clause. If there is no <else case>, execution continues with execution of the next statement following the ENDCASE statement. If control is transferred to the <else case>, execution continues with execution of the first executable statement following the ELSECASE until the ENDCASE statement is reached.

If the value of the <case-heading> <expression> matches one of the known constant values of a <case prefix> <expression>, control is transferred to the matched <case prefix> statement. Execution continues with execution of the first statement of the <case section> statement sequence until a subsequent <case prefix>, an <else case>, or an ENDCASE statement is reached. Execution will then continue with execution of the next statement following the DO CASE-group's ENDCASE statement.

6.3.3 Transfer Relative to a DO CASE-group

Transfer of control into a DO CASE-group is not permitted. Transfer of control to a DO CASE-group must include execution of the <case-heading> statement. Transfer of control out of a <group> via a <goto statement> is not permitted.

6.2.6 ENDDO Statement Processing

Execution of the ENDDO statement occurs as a result of the normal execution sequence. Execution continues with loop control processing (6.2.4) of the same DO WHILE-loop.

6.3 The Do Case Group

The DO CASE-group specifies which one of several possible program elements is to be executed next, depending on the value of a given arithmetic expression.

Syntax:

```

<case group> ::= <case heading> <case body> <endcase>
<case heading> ::= DO CASE <expression> ;
<case body> ::= <case section>... [ <else case> ]
<case section> ::= <case prefix>... <basic statement>...
<case prefix> ::= CASE ( <cvi expression>
                        [ : <cvi expression> ] ) :
<else case> ::= ELSECASE <basic statement>...
<endcase> ::= ENDCASE [ <identifier> ] ;

```

All occurrences of <expression> must be of type <cardinal> or <fixed> scaled zero. Furthermore, the values of <cvi expression> in all <case prefix>'s must be mutually exclusive.

If two <cvi expression>s, separated by a colon, are specified in a <case prefix>, they represent a range of values, i.e., the first <cvi expression> must have a value less than the second <cvi expression>. In this option all integer values in the range will select the <basic statement> prefixed by it.

If the optional <identifier> is specified on the ENDCASE statement, then the corresponding DO CASE statement must have a <label prefix> identical to <identifier>.

A typical example of a DO CASE-group might be:

```

DECLARE ( i, j ) CARDINAL(16);
DO CASE i;
CASE(0): j = 10;
CASE(1):
CASE(3:4): j = 7;
ELSECASE
j = 3;
ENDCASE;

```

6.2.2 Active and Inactive DO WHILE-loops

A DO WHILE-loop is either active or inactive. Initially inactive, a DO WHILE-loop becomes active only when its DO WHILE statement is executed.

Once active, the DO WHILE-loop becomes inactive only when:

- (1) the evaluation of <expression> yields all zero bits
- (2) a RETURN statement is executed within the range of the loop
- (3) an EXIT statement is executed within the range of the loop that identifies this loop to be terminated.

6.2.3 Executing a DO WHILE Statement

The effect of executing a DO WHILE statement is to evaluate the <expression> producing a value of type bit-string result.

At the completion of execution of the DO WHILE statement, loop control processing begins.

6.2.4 Loop Control Processing

Loop control processing determines if execution of the range of the DO WHILE-loop is required. The <expression> is evaluated; the resultant value must be a scalar of type bit-string. If the bit-string result contains any one bits, execution of the first statement in the range of the DO WHILE-loop begins. If the bit-string result of the <expression> evaluation contains only zero bit values, the DO WHILE-loop becomes inactive, and normal execution continues with execution of the next executable statement following the range of the DO WHILE.

6.2.5 Execution of the Range

Statements in the range of a DO WHILE-loop are executed until the corresponding ENDDO statement is reached. The value of <expression> should be redefined during execution of the range of the DO WHILE-loop.

Transfer of control out of a <group> via a <goto statement> is not permitted.

After execution of the above statements, I=11, J=10, K=6, L=5, and N=50.

Also consider the following example:

```

DECLARE ( i, j, k, l, n ) CARDINAL(16);
n = 0;
DO i = 1 TO 10;
  j = 1;
  DO k = 5 TO 1 BY 1;
    l = k;
    n = n + 1;
  ENDDO;
ENDDO;

```

After execution of the above statements, I=11, J=10, K=5, and N=0. L is not defined by the above statements.

6.2 Repetitive DO-Loop

The range of the repetitive DO WHILE-loop is executed as long as the bit-string result of evaluating the scalar <expression> contains at least one true (1 bit) value.

Syntax:

```

<repetitive loop> ::= <do while> <group body>
                        ENDDO [ <identifier> ] ;
<do while> ::= DO WHILE ( <expression> ) ;
<group body> ::= { [ <label prefix> ]
                    <basic statement> }...

```

The evaluation of <expression> in <do while> must yield a scalar value of type bit-string. If the optional <identifier> is specified on the ENDDO statement, then the corresponding DO WHILE statement must have a <label prefix> identical to <identifier>.

6.2.1 Range of a DO WHILE-loop

The range of a DO WHILE-loop consists of the executable statements from and including the first executable <basic statement> following the DO WHILE statement that specifies the DO WHILE-loop to and including the corresponding ENDDO statement of the DO WHILE-loop. Transfer of control into the range of a DO WHILE-loop is not permitted.

6.1.5 Execution of the Range

Statements in the range of a DO-loop are executed until the corresponding ENDDO statement is reached.

Transfer of control out of a <group> via a <goto statement> is not permitted.

6.1.6 ENDDO Statement Processing

Execution of the ENDDO statement occurs as a result of the normal execution sequence and initiates incrementation processing as described below. If the optional <identifier> is specified on the ENDDO statement, then the corresponding DO statement must have a <label prefix> identical to <identifier>.

6.1.7 Incrementation Processing

Incrementation processing has the effect of the following steps performed in sequence:

- (1) The <index> and the incrementation parameter m3 of the DO-loop are selected for processing.
- (2) The value of the <index> is incremented by the value of m3 as though the assignment statement

<index> = <index> + m3;

were executed.

- (3) Execution continues with the loop control processing (6.1.4) of the same DO-loop.

An example illustrates the above:

```

DECLARE ( i, j, k, l, n ) CARDINAL(16);
n = 0;
DO i = 1 TO 10;
  j = 1;
  DO k = 1 TO 5;
    l = k;
    n = n + 1;
  ENDDO;
ENDDO;

```

8.4.5.3 From Option

The <from option> must be used in the <write statement> for a file with the OUTPUT attribute or a file with the DIRECT and UPDATE attributes. It must be used in the <rewrite statement> for a file with the UPDATE attribute. The <from option> specifies the variable from which the record is written.

Syntax:

<from option> ::= FROM(<reference>)

The reference is a variable in connected storage that indicates the source of the record to be written.

8.4.5.4 Ignore Option

The <ignore option> can be used in a <read statement> for a direct or consecutive file organization with the SEQUENTIAL and INPUT attributes or a direct file organization with the UPDATE attribute.

Syntax:

<ignore option> ::= IGNORE(<expression>)

The expression value specifies the number of records to be skipped or ignored.

8.4.5.5 Key Option

The <key option> applies to a direct file organization with the DIRECT attribute. When the DIRECT attribute is specified the KEYED attribute is assumed. The <key option> identifies a particular record. It may be used in a <read statement> for a file with the DIRECT and INPUT attributes or for a file with the DIRECT and UPDATE attributes. It is also used in a <rewrite statement> or <delete statement> for a file with the DIRECT and UPDATE attributes.

Syntax:

<key option> ::= KEY(<expression>)

The expression value will be a fixed-point integer value or a <cardinal> value that represents the key number of the record to be accessed.

- 4) For DIRECT or SEQUENTIAL access mode, the record being deleted must exist.

8.4.5 Transmission Options

Options that are permitted for record transmission statements are:

<file option>
<into option>
<from option>
<ignore option>
<key option>
<keyfrom option>

Each option consists of a key word followed by a <reference> or <expression>. In a statement options may appear in any order.

8.4.5.1 File Option

The <file option> must appear in every input/output transmission statement. It specifies the name of the file that is being acted upon.

Syntax:

<file option> ::= FILE(<reference>)

The <reference> specifies the name of the file.

8.4.5.2 Into Option

The <into option> must be used in the <read statement> for a file with the INPUT or UPDATE attribute. The <into option> specifies the variable to which the logical record is assigned.

Syntax:

<into option> ::= INTO(<reference>)

The <reference> is a variable in connected storage that specifies the destination of the record being read.

3) For SEQUENTIAL access mode, the <rewrite statement> replaces the last record read.

4) For DIRECT access mode, the <rewrite statement> will replace an existing record whether or not it has first been read.

8.4.4 The Delete Statement

The <delete statement> can be used only with a file having the UPDATE attribute. It deletes an existing record from a dataset.

Syntax:

```
<delete statement> ::= DELETE <file option>
                        [ <key option> ] ;
```

8.4.4.1 Delete Options

1) The <file option> is required and specifies the file where the record deletion is to take place. If the file has not been opened explicitly by an <open statement> it will be opened. The default open attributes are UPDATE and DIRECT.

2) The <key option> specifies the key of the record that is to be deleted. The expression is evaluated to a <cardinal> or fixed-point integer value. The <key option> must be specified if the file has DIRECT and UPDATE attributes; it cannot be specified otherwise.

8.4.4.2 General Rules for the Delete Statement

- 1) The function mode must be UPDATE.
- 2) The access mode may be DIRECT or SEQUENTIAL.

If the access mode is DIRECT, then:

- a) The <keyed attribute> is implied if not specified.
- b) The <key option> is required.
- c) The file organization must be direct.

If the access mode is SEQUENTIAL, then:

- a) The <key option> is not permitted.
- b) The file organization may be consecutive or direct.

- 3) For SEQUENTIAL access mode, the record deleted will be the last one read.

4) If the <write statement> is used for a file having DIRECT and UPDATE attributes, the record whose key is specified must not already exist in the file.

8.4.3 The Rewrite Statement

The <rewrite statement> can be used only with a file having the UPDATE attribute. It replaces an existing record in a dataset.

Syntax:

```
<rewrite statement> ::= REWRITE <file option> <from option>  
                        [ <key option> ] ;
```

8.4.3.1 Rewrite Options

1) The <file option> is required; it specifies the file to which a record is transmitted. If the file has not been opened explicitly by an <open statement> it will be opened. The default open attributes are UPDATE and DIRECT.

2) The <from option> is required. It specifies a variable in connected storage from which the record replacing the existing record is to be written.

3) The <key option> is required for a direct file organization; it specifies the position of the record that is to be written. The expression must evaluate to a fixed-point integer value or <cardinal> value.

8.4.3.2 General Rules for the Rewrite Statement

1) The function attribute must be UPDATE.

2) The access mode may be DIRECT or SEQUENTIAL

If the access mode is DIRECT then:

- a) The <keyed attribute> is implied if not specified.
- b) The <from option> and <key option> are required.
- c) The file organization must be direct.

If the access mode is SEQUENTIAL then:

- a) The <key option> is not permitted.
- b) The <from option> must be specified.
- c) The file organization may be consecutive or direct.

8.4.2 The Write Statement

The <write statement> can be used by a file with the <output attribute> or the <update attribute>. It causes a record to be transmitted from a variable in connected storage to the associated dataset.

Syntax:

```
<write statement> ::= WRITE <file option> <from option>  
                        [ <keyfrom option> ] ;
```

8.4.2.1 Write Options

1) The <file option> is required. It specifies the file to which a record is transmitted. If the file has not been opened by an explicit <open statement>, it will be opened. The default open attributes are OUTPUT and SEQUENTIAL.

2) The <from option> is required; it specifies the variable in connected storage from which the record is written.

3) The <keyfrom option> specifies the position of the record that is to be written. The <expression> is evaluated to a <cardinal> or fixed-point integer value. This option is required with a direct file organization.

8.4.2.2 General rules for the Write Statement

- 1) The function attribute must be OUTPUT or UPDATE.
- 2) The access mode may be DIRECT or SEQUENTIAL.

If the access mode is DIRECT then:

- a) The <keyed attribute> is implied if not already specified.
- b) The <from option> and <keyfrom option> are required.
- c) The file organization must be direct.

If the access mode is SEQUENTIAL then:

- a) The function mode must be OUTPUT.
- b) The <from option> is required.
- c) The <keyfrom option> is not permitted.
- d) The file organization must be consecutive.

3) The <write statement> cannot be used with a file having SEQUENTIAL and UPDATE attributes. The <rewrite statement> should be used in this case.

Syntax:

```

<read statement> ::= READ <file option>
                    { <into option> | <ignore option> }
                    [ <key option> ] ;

```

8.4.1.1 Read Options

1) The <file option> is required; it specifies the file-name of the file from which the record is read. If this file has not been opened by an explicit <open statement> it will be opened. The default attributes are INPUT and SEQUENTIAL.

2) The <into option> specifies the variable into which the record is to be read. The reference must specify a variable in connected storage.

3) The <key option> specifies the position of the record that is to be read. The <key option> is used only with the direct file organization. The expression is evaluated to a <cardinal> or fixed-point integer value.

4) The <ignore option> specifies that records are to be skipped. The value of <expression> specifies how many records are to be skipped.

8.4.1.2 General Rules for the Read Statement

- 1) The function attribute must be INPUT or UPDATE.
- 2) The access mode may be DIRECT or SEQUENTIAL

If the access mode is DIRECT then:

- a) The <keyed attribute> is implied, if not already specified.
- b) The <key option> and <into option>s are required.
- c) The <ignore option> is not permitted.
- d) The file organization must be direct.

If the access mode is SEQUENTIAL then:

- a) Either the <into option> or the <ignore option> is required. If the <ignore option> is specified, the number of records to be skipped must be specified.
- b) The <key option> is not permitted.
- c) The file organization may be consecutive or direct.

8.3.5 The Close Statement

The <close statement> applies to a file. A file that is not closed explicitly by a close statement is closed at the termination of program execution.

Syntax:

```
<close statement> ::= CLOSE <single close>
                        [, <single close> ] ;
<single close> ::= <file option>
                  <environment>
```

8.3.5.1 Close Attributes

1) The <file option> is required, it specifies the file name of the file being closed. The file <reference> must be the same as the one referenced in the file <open statement>. More than one file may be closed with one <close statement> by repeating <single close> for each file. The statement requires a comma separating each pair of <single close>s.

2) The <environment> option on a <close statement> provides a means of indicating the disposition of the file at the time it is closed, such as: rewind, leave, etc. It also indicates what computer resources may be released. If both the <open statement> and <close statement>s for a file contain file dispositions, the disposition on the <close statement> takes precedence.

8.4 Data Transmission statements

8.4.1 The Read Statement

The <read statement> can be used by a file with the <input attribute> or <update attribute>. It causes one record to be transmitted from the associated dataset to a variable in connected storage defined in the program. It may also be used with the <ignore option> to skip a specified number of records.

2) The <record attribute> is required if not declared with the file. It specifies that each record is transmitted as an entity.

3) Only one of the attributes INPUT, OUTPUT, UPDATE may appear in an <open statement>. If all three are omitted, INPUT is assumed.

The <input attribute> specifies that the file will be used to transmit data from an input source to memory.

The <output attribute> specifies that a new dataset is to be created, and that the data will be transmitted from memory to that dataset.

The <update attribute> specifies that data will be transmitted to and from a dataset which can function as both input and output. The <update attribute> permits the program to read and write the same file or the same record within a file.

4) The <direct attribute> or <sequential attribute> define the files access mode. Either <direct attribute> or <sequential attribute> may appear in the <open statement>, not both. If neither appears the default assumed is SEQUENTIAL.

The <direct attribute> implies that the program will exercise direct control over where the next record will be read or written. The program must supply a key whose value indicates the location of the record in the file.

The <sequential attribute> implies that the records will be read or written in consecutive order. This means in physical order for a consecutive file organization and in relative key order for a direct file organization.

5) Direct files must have a <keyed attribute>. If the <keyed attribute> is specified, the record transmission statements can refer to a record using the <key option> or <keyfrom option>.

6) The <title option> is optional and specifies an external file-name.

8.3.4 Closing a File

Closing a file disassociates a named file from a dataset and terminates the input/output process. It also releases computer resources required during the input/output process. A file must be closed before it can be associated with another dataset.

<declare statement> and the explicit <open statement> are merged. The same attributes need not be specified in both statements but there must be no conflicts.

An implicit file opening will occur when an input/output statement is executed for a file that is not already opened. In this case an implied <open statement> is assumed. The type of input/output statement determines the default file attributes of the <open statement>.

The following table indicates The default attributes associated withThe various input/output transmission statements.

<u>statement</u>	<u>default file attributes</u>
READ	INPUT, SEQUENTIAL
WRITE	OUTPUT, SEQUENTIAL
DELETE	UPDATE, DIRECT
REWRITE	UPDATE, DIRECT

In the case of an implicit open, the DECLARE attributes, if any, will take precedence over the default attributes implied in the implicit open.

8.3.3 The Open Statement

The <open statement> always applies to a file. All files must be opened before data in the file can be read or written.

Syntax:

```

<open statement> ::= OPEN <single open>
                        [, <single open> ] ;

<single open> ::= <file option>
                { [ RECORD ]
                  ■ [ <title option> ]
                  ■ { INPUT | OUTPUT | UPDATE }
                  ■ { SEQUENTIAL | DIRECT [ KEYED ] }
                  ■ [ <environment> ] }

<title option> ::= TITLE( <expression> )

```

8.3.3.1 Open Attributes

1) The <file option> is required ; it specifies the file-name of the file that is being opened.

The <direct attribute> specifies that a record in a file is accessed on the basis of its location in the file. The location of the record is determined by a key. Therefore the <direct attribute> implies the <keyed attribute>. The associated dataset must be on a direct access device. The <direct attribute> is required in the <open statement> associated with a file that is to be accessed randomly.

8.3.1.8 Keyed

Syntax:

<keyed attribute> ::= KEYED

The <keyed attribute> indicates that the records in the file are to be accessed using a key option (KEY,KEYFROM) in the data transmission statements. The

<keyed attribute> is assumed if the <direct attribute> is specified in the <open statement>.

8.3.1.9 Environment

Syntax:

<environment> ::= ENVIRONMENT(<option list>)

The <environment> attribute provides information that permits the computer to determine the method of accessing data associated with a file. It specifies the physical organization of the dataset associated with the file and how the dataset is handled.

The <option list> of the <environment> attribute is compiler implementation dependent, therefore the detailed description is not covered in this language specification document. The

<option list> may specify such attributes as: dataset file organization; record size; block size; device type; open status; and the disposition of the file at the time it is closed.

8.3.2 Opening a file

Opening a file associates a file name with a dataset. It also defines attributes associated with the file if they have not been defined in a <declare statement>.

A file may be opened explicitly with the <open statement>. If the explicit <open statement> is present, it must appear before any input/output transmission statements on that file. Attributes specified in the

8.3.1.4 Output

Syntax:

<output attribute> ::= OUTPUT

The <output attribute> is one of three attributes which determine the direction of data transmission. The <output attribute> is a required attribute in the <open statement> associated with a file that is to be written only.

8.3.1.5 Update

Syntax:

<update attribute> ::= UPDATE

The <update attribute> is one of three attributes which determine the direction of data transmission. The <update attribute> indicates that the associated file can be read or written. It permits records to be inserted into an existing file and permits altering or deleting existing records. The <update attribute> is required in the <open statement> associated with a file that is being updated (i.e. modified by a <delete statement> or a <rewrite statement>).

8.3.1.6 Sequential

Syntax:

<sequential attribute> ::= SEQUENTIAL

The <sequential attribute> specifies that successive records in a file are to be accessed on the basis of their consecutive physical positions. In other words they must be read in the exact order that they were written. The <sequential attribute> is required in the

<open statement> associated with a file that is to be written or accessed sequentially.

8.3.1.7 Direct

Syntax:

<direct attribute> ::= DIRECT

The file attributes that apply to record input/output are as follows:

```

<file attribute>
<record attribute>
<input attribute>
<output attribute>
<update attribute>
<sequential attribute>
<direct attribute>
<keyed attribute>
<environment>( <option list>)

```

The attributes are members of an <attribute set> in the <declare statement> or the <open statement>.

8.3.1.1 File

Syntax:

```
<file attribute> ::= FILE
```

The <file attribute> in the <declare statement> specifies that the associated <reference> is a file variable. The <file attribute> must be specified in the <declare statement>, unless one of the other file attributes is specified which implies the <file attribute>.

8.3.1.2 Record

Syntax:

```
<record attribute> ::= RECORD
```

The <record attribute> causes a file to be treated as a sequence of records, each record consisting of one or more data items recorded in any internal form. It is required in the <open statement> or the <declare statement>.

8.3.1.3 Input

Syntax:

```
<input attribute> ::= INPUT
```

The <input attribute> is one of the three attributes which determines the direction of data transmission. INPUT applies only to files that are to be read. It is required in the <open statement> associated with a file that is to be a read only file.

8.2.1 Dataset Organization

A dataset is a collection of data external to the program. A distinct physical grouping of data on an input/output medium is defined as a physical record which is the unit of data transferred between main storage and input/output devices.

Two kinds of dataset organizations, consecutive and direct are defined. In a dataset with consecutive organization, records are sequenced on the basis of their physical position. The records have no keys. The consecutive dataset can only be accessed sequentially. In a dataset with direct organization, the records have a relative position within the dataset. Each record has a key which defines the position number of the record. A direct dataset may be accessed sequentially or randomly based on a key.

Record oriented datasets are operated upon by the execution of a <read statement>, <write statement>, <rewrite statement>, and a <delete statement>.

8.2.2 Consecutive Dataset

A consecutive dataset is an ordered sequence of records without keys. A consecutive dataset must be accessed in the same order in which it was written.

8.2.3 Direct dataset

A Direct dataset is an ordered set of records, each identified by a unique key. The location of a record in a direct dataset is determined by the value of a key. The associated dataset must be on a direct-access device.

8.3 Files

A file is defined as a group of logical records which may be regarded independent of the physical properties of a dataset and the physical medium on which it is stored. The language defines a file by a filename and a set of file attributes. A file-name is a <reference>.

8.3.1 File Attributes

8. INPUT/OUTPUT

8.1 Introduction

This language includes input and output statements which enable data to be transmitted between internal storage and peripherals. A collection of data external to a program is called a dataset. Transmission of data from a dataset to internal storage is called input, and transmission of data from internal storage to a dataset is called output.

Input and output statements, from a language viewpoint, are associated with the logical organization of a dataset and not with its physical(environment) characteristics. The Input/Output statements can be defined without specific knowledge of the physical characteristics of the peripheral devices that will be utilized.

To permit a source program to consider the logical aspects of data rather than the physical characteristics in a dataset, the language will provide a symbolic representation of a dataset called a file. A file can be associated with different datasets during the execution of a program.

In this language record-oriented input/output will be supported. In record-oriented transmission, the data is considered to be a collection of discrete records. No data conversion takes place during the record transmission.

The advantages of record-oriented transmission are:

- 1) more efficient execution since no conversions are performed
- 2) less space is required on external storage devices
- 3) any format is acceptable
- 4) programmer has more control and awareness of the structure of the data.

This chapter will cover the syntax and semantics of record-oriented Input/Output.

8.2 Datasets

Examples:

Allocate an array dynamically using the current value of *x* for the extent of its dimensions.

```
declare A (X, X) bit(32) controlled ;
```

```
    X = 16;
    allocate A ;
```

7.3 FREE Statement

Syntax:

```
<free statement> ::= FREE <reference>
                    [ , <reference> ] ... ;
```

The FREE statement releases storage from association with a variable having the CONTROLLED attribute. When the storage is freed, it can be used for other variables.

The "reference" must be an unsubscripted level-one variable. If it has no storage allocated to it when the FREE statement is executed, no attempt is made to free it.

Examples:

Allocate an array B with two different extents at different points during execution of the program.

```
declare B(X, X) fixed(31, 0) controlled;
```

```
    X = 8;
    allocate B;
    :
    :
    free B;
    X = 16;
    allocate B;
    :
    :
    free B;
```

The combined use of the ALLOCATE and FREE statements can facilitate more efficient use of storage in certain situations. If many small allocations are needed, prudent usage of ALLOCATE and FREE can minimize fragmentation. For example, the size of such variables in one program could all be multiples of each other or all multiples of powers of two.

Whenever the <target> reference is an array, and the source <expression> value is a scalar, the scalar value is replicated and stored into all specified elements of the <target> reference array.

7.1.4 Structure Assignment

Structure assignment applies when a <target> reference is a <structure>. If the source <expression> value is a scalar, it is assigned to each member of the <target> structure according to the rules of scalar assignment.

If the source <expression> value is a <structure>, the source <expression> and <target> reference must have conforming structure for corresponding containing levels. The number and structure of containing members must match on each level. Structure members from the <expression> are promoted to an aggregate value that conforms with the corresponding level of the <target> reference and are converted, where allowed, to the <target> member data type. The possibly promoted and converted value is assigned to the <target> member. The promotions and conversions do not effect the value of the source <expression>.

As is the case for array assignment, the order of assignment of individual data items in structure assignment is not specified and any assignment which depends on the order is in error.

7.2 ALLOCATE Statement

Syntax:

```
<allocate statement> ::= ALLOCATE <allocate list> ;
<allocate list> ::= <allocate reference>
                    [ , <allocate reference> ] ...
<allocate reference> ::= <identifier>
```

The ALLOCATE statement allows control of association of a dynamic variable (one having the attribute CONTROLLED) with storage. The execution of the statement causes storage to be allocated for a specified variable or list of variables.

The amount of storage to be allocated, called the "extent" of the variable, is supplied in the declaration of the variable and can be non-constant in value. If insufficient storage exists within "system storage", the STORAGE condition occurs(see section 9).

The variables allocated remain so when control leaves the block in which the allocation was performed. The variables are dis-associated from storage by the FREE statement.

(i+j-1)-th character or bit. All other characters or bits of the string are unmodified.

7.1.2 Array Assignment

If the <target> is an array reference, array assignment is performed to each of the elements of the reference. Various cases apply according to the aggregate type of the source <expression>. If the <expression> is a scalar, then a replication is performed as described in the next section.

If the source <expression> is an array the value of the <expression> and the <target> reference must have the same number of dimensions. The assignment is performed by doing scalar assignments on an element-by-element basis in each dimension using identical subscript values for the source and target.

The order in which the assignment to individual elements is done is deliberately left unspecified. Any assignment in which the order of assignment to array elements is significant is incorrect and the program is in error.

For each dimension the rule of "the lesser of the two corresponding extents" applies (see also section 4.6 on "Conformance") when assigning the array element values to the corresponding items of the <target> reference. The rule is defined as two cases which apply to each dimension of the arrays below.

Case 1: The given dimension of the <target> has a smaller number of elements than the corresponding dimension of the source <expression> array.

Result: The assignment of elements is complete when the extent of the <target> reference is reached. Also alignment of array elements occurs before the assignments are made. Unused elements in the source array <expression> do not participate in the assignment operation.

Case 2: The given dimension of the <target> has a larger number of elements than the corresponding dimension of the source array <expression>.

Result: The assignment of elements is complete when the extent of the source array <expression> is reached. Unused elements in the <target> array do not participate in the assignment operation.

7.1.3 Scalar Broadcast Assignment

7.1.1.2 String Assignment

If the <target> is of type string, string assignment is performed.

If the source <expression> is a <reference> to a scalar character-string or bit-string variable, the generation of storage identified by that <reference> cannot overlap the generation identified by the <target> reference, unless the <target> and the source <reference> identify exactly the same generation of storage.

For a <target> with the <varying> attribute the length of the source <expression> string must not exceed the declared maximum length for the <target>. If it does, the STRINGSIZE condition is signalled (see section 9 for conditions) and the assignment does not occur. If the length is not greater than the declared <target> maximum length, the source <expression> string value is assigned to the <target> reference. The length of the <expression> becomes the new length of the <target> string value.

For a <target> without the <varying> attribute, if the source <expression> string value is longer than the declared length of the <target> reference then it is truncated on the right to the length of the <target> string. If the <expression> is shorter it is padded on the right with ASCII blanks to the length of the <target> string.

7.1.1.3 Arithmetic Assignment

The source <expression> is assigned to the <target> according to the precision rules for infix operations as defined in section 4.4.

7.1.1.4 Substr Pseudo Variable

If the <target> is the <substr pseudo>, the values of the <expression>s must be either fixed-point integers or of type <cardinal>. Let i be the value of the first <expression> and j be the value of the second <expression>. If the string variable identified by the <reference> is declared with the <varying> attribute, let n be the current length of the string variable value; otherwise, let n be the evaluated string length associated with the variable's generation of storage.

If the second <expression> is omitted, let j be $n-i+1$. If $(0 \leq i-1 \leq j+i-1 \leq n)$ is not satisfied, the STRINGRANGE condition occurs. If detection of the condition is disabled, the program is in error and the results of continued execution are undefined.

If the inequality is satisfied, the value is assigned to the string variable beginning with the i -th character or bit and continuing through the

7. ASSIGNMENT AND ALLOCATION STATEMENTS

7.1 Assignment Statement

The <assignment> statement is the primary facility for the manipulation of data. An <assignment> statement consists of two parts connected by the replace "=" operator. The left-hand side of the statement is called the <target> and is either a <reference> or a <pseudo variable>. The right-hand side is called the source <expression>.

Syntax:

```
<assignment statement> ::= <target> = <expression> ;
<target> ::= <reference> | <pseudo-variable>
<pseudo-variable> ::= <substr pseudo>
<substr pseudo> ::= substr ( <reference>
                             , <expression> [ , <expression> ] )
```

The execution of an assignment statement results in the <target> reference becoming defined with the value of the source <expression>.

7.1.1 Scalar Assignment

An <assignment> statement is applied by evaluating the source <expression> and then assigning the resultant value to the <target> reference.

If the <target> reference is a scalar, the assignment is scalar and the current value of the <target> reference is replaced by the possibly promoted and converted value of the <expression> on the right.

7.1.1.1 Data Type Conversion

If the source <expression> and <target> are both of either data type <fixed> or <cardinal>, conversion is performed according to the type of the <target>.

Except for the fixed-point types <fixed> and <cardinal>, the data type of the source <expression> value must be the same as the data type of the <target> reference. There is no other implicit conversion from one data type to another. Such conversions must be made by the explicit use of built-in functions.

5.3 for further information on multiple-entry procedures.

6.7 EXIT Statement

The EXIT statement is used to terminate execution within a <group>.

Syntax:

```
<exit statement> ::= EXIT [ <identifier> ] ;
```

where <identifier> is a statement label and must be the same as a statement label on a DO CASE, DO, or DO WHILE, that physically precedes the EXIT statement. An EXIT statement which names a label must be contained within the <group> with the same label. An example is if the label of the EXIT references a DO WHILE statement label, then execution continues with execution of the first executable statement following the ENDDO of the DO WHILE loop.

Control is returned to the statement following the CALL statement when the invoked procedure is terminated by executing a RETURN statement or by passing through the ROUTINE statement or ENDPROC statement delimiting the scope of the procedure body.

6.5.2 RETURN Statement

Execution of a RETURN statement terminates the reference of a procedure.

Syntax:

```
<return statement> ::= RETURN [ ( <function value> ) ] ;
<function value> ::= <expression>
```

Such procedures may contain more than one RETURN statement; however, a procedure need not contain a RETURN statement. Execution of an ENDPROC statement in a procedure has the same effect as executing a RETURN statement in the procedure.

The <function value> is allowed only for a function and may be a scalar or an aggregate.

6.6 GO TO Statement

The GO TO statement is used to transfer control to another statement.

Syntax:

```
<goto statement> ::= GO TO <identifier> ;
                   | GOTO <identifier> ;
```

The <identifier> is the statement label of an executable statement written as a <label prefix> in the same <procedure> as the <goto statement>. The <goto statement> transfers control to another <statement> on the same structural level of the procedure. The transfer of control indicated by the <goto statement> may not be from outside the body of a <group> or <conditional statement> into the body of the <group> or <conditional statement> respectively, nor the opposite (outward) direction.

There is one exception; a <goto statement> may be written in an <on unit> to terminate its activation and resume program execution at another statement.

Within a <procedure> the <identifier> must be defined in the same <procedure component> as the occurrence of the <goto statement>. See section

Example 3.

Let bit-string E contain only false values.

After execution of the above statements, A[i] will be set equal to C[i]+D[i] for all i, and D[i] is set equal to A[i] for all i. The two statements between the THEN keyword and ELSE statements are not executed.

6.4.3 The Logical If Statement

Execution of the <logical if statement> causes evaluation of the <expression> producing a scalar value of type bit-string result.

6.4.3.1 Logical If Statement Flow of Control Processing

Once the <expression> has been evaluated, the scalar bit-string result is examined for any one bits. If there exists at least one, the <single statement> in <logical if statement> is executed. If there are no one bits, the execution sequence continues with the next statement after the <logical if> statement.

6.5 CALL and RETURN Statements

Execution of a CALL statement causes control to be transferred to the specified procedure.

Syntax:

```
<call statement> ::= CALL <entry reference>
                    [ <argument list> ] ;
```

6.5.1 CALL Statement

The CALL statement causes the entry value corresponding to the named <entry reference> to be invoked. Invoking an entry value causes the associated procedure to be activated in the environment of the CALL statement and control to be transferred to the start of the procedure.

The number of <expression>s in the <argument list> of the <entry reference> must be equal to the number of <identifier>s in the <parameter list> of the <procedure statement>, <routine statement>, or <entry> declaration identified by the value of the <entry reference> in the <call statement>. A CALL statement is executed by evaluating the <entry reference> and all <expression>s in an unspecified order.

If detection of the condition identified by the <condition name> is disabled, transfer control to the next statement; otherwise signal the condition. A signal for a condition causes the most recently established <on unit> for the condition to be invoked as a <procedure>. See section 5 for flow of control with respect to <on unit>s and <procedure>s.

9.2.2 The Revert Statement

The <revert statement> is executed by evaluating the <condition name>s in any order. For each <condition name>, disassociate any <on unit> established by the current block activation. This has the effect of resetting the association to its state upon entry to the current block.

9.1.16 Underflow Condition

The UNDERFLOW condition occurs when the result of a floating-point computation has a binary exponent less than -128.

9.1.17 Zerodivide Condition

The ZERODIVIDE condition occurs when division is attempted with a divisor of zero.

9.2 Statements and Program Effects

The detection of a condition is called a signal. A signal results in activation of the most recently established <on-unit> for the condition. An <on-unit> is established by the execution of an <on statement> and is reverted by the execution of a <revert statement>. Only one <on-unit> may be established for each condition in a procedure. Only those <on-unit>s established in a given <procedure> can be reverted in the same <procedure>.

Syntax:

```
<on statement> ::= ON <condition name> <on-unit>
<on-unit> ::= <statement list> ENDON;
           | [SYSTEM] ;
```

```
<revert statement> ::= REVERT <condition name>
                    [ ,<condition name>]... ;
```

```
<signal statement> ::= SIGNAL <condition name>;
```

If the SYSTEM <on-unit> is used, or if no <on-unit> has been established for a condition, the default system <on-unit> for the condition is invoked if the condition is signalled.

The form

```
ON <condition name> ;
```

disables the <condition name> for the execution of the current block.

9.2.1 The Signal Statement

The execution of a <signal statement> signals the condition indicated by the <condition name>.

9.1.9 Size Condition

The SIZE condition occurs when the result of a fixed-point computation exceeds the precision of the target.

9.1.10 Storage Condition

The STORAGE condition occurs when the storage used for allocation of variables, primarily those of dynamic and automatic storage class, is full.

9.1.11 Stringrange Condition

The STRINGRANGE condition occurs when the built-in function 'substr' or the <substr pseudo> specify a substring that exceeds the range of the given string.

9.1.12 Stringsize Condition

The STRINGSIZE condition occurs when a value is converted to a string data type that does not have sufficient length to represent the value.

9.1.13 Subscriptrange Condition

The SUBSCRIPTRANGE condition occurs when the value of a subscript or <selection qualifier> exceeds the <extent> of the dimension to which it applies.

9.1.14 Transmit Condition

The TRANSMIT condition occurs when an unrecoverable transmission error is detected during an I/O operation on the file specified by <reference>.

9.1.15 Undefinedfile Condition

The UNDEFINEDFILE condition occurs during an OPEN when the file specified by <reference> cannot be associated with any file in the system.

9.1.3 Error Condition

The ERROR condition is signalled by the default <on-unit>s and by the built-in functions. The standard system action is to post an implementation defined message about the error and then raise the FINISH condition.

9.1.4 Finish Condition

The FINISH condition terminates execution of a program and is raised when the ENDPROC of the initial <external procedure> is reached.

It can also be raised by a SIGNAL statement or the standard system action for the ERROR condition.

9.1.5 Fixedoverflow Condition

The FIXEDOVERFLOW condition occurs when the result of a binary fixed-point computation exceeds the implementation precision limit of 63 for <fixed> and 64 for <cardinal> data.

9.1.6 Key Condition

The KEY condition occurs during the execution of a <read statement> <rewrite statement>, or a <delete statement>. It results when an invalid key value is encountered or when the specified key cannot be matched to any record in the file specified by the <reference>. The KEY condition can also occur during the execution of a <write statement> if an invalid key is encountered or if a record already exists with the same key value.

9.1.7 Overflow Condition

The OVERFLOW condition occurs when the result of a floating point computation has a binary exponent that exceeds 127.

9.1.8 Record Condition

The RECORD condition occurs during the execution of a <read statement> or <rewrite statement> if the record size declared for the file specified by <reference> is not equal to the size of the <reference> variable in the <read statement> or <rewrite statement>.

9. CONDITIONS

9.1 Conditions in the Language

During the execution of a program certain events can occur which prevent further processing or require special action to allow processing to continue. The occurrence of such an event is called a condition. Each condition which can be detected during execution is given a condition name to identify it.

The language provides the means to control the processing which follows the detection of a condition.

The conditions in the language are:

Syntax:

```
<condition name> ::= CONVERSION
| ENDFILE ( <reference> )
| ERROR | FINISH
| FIXEDOVERFLOW
| KEY ( <reference> )
| OVERFLOW
| RECORD ( <reference> )
| SIZE | STORAGE | STRINGRANGE
| STRINGSIZE | SUBSCRIPTRANGE
| TRANSMIT ( <reference> )
| UNDEFINEDFILE ( <reference> )
| UNDERFLOW | ZERODIVIDE
```

9.1.1 Conversion Condition

The CONVERSION condition occurs when an invalid character-string is converted to an arithmetic data type.

9.1.2 Endfile Condition

The ENDFILE condition occurs when an end-of-file is encountered on the file specified by the <reference> during execution of a <read statement>.

8.4.6.6 I/O Transmission Statement for direct Datasets

Access Mode	Transmission Statement
SEQUENTIAL INPUT	READ FILE(<reference>)INTO(<reference>); READ FILE(<reference>)IGNORE(<expression>);
SEQUENTIAL UPDATE	READ FILE(<reference>)INTO(<reference>); READ FILE(<reference>)IGNORE(<reference>); REWRITE FILE(<reference>)FROM(<reference>); DELETE FILE(<reference>);
DIRECT INPUT	READ FILE(<reference>)INTO(<reference>) KEY(<expression>);
DIRECT OUTPUT	WRITE FILE(<reference>)FROM(<reference>) KEYFROM(<expression>);
DIRECT UPDATE	READ FILE(<reference>)INTO(<reference>) KEY(<expression>); WRITE FILE(<reference>)FROM(<reference>) KEYFROM(<expression>); REWRITE FILE(<reference>)FROM(<reference>) KEY(<expression>); DELETE FILE(<reference>)KEY(<expression>);

8.4.6.5 I/O Transmission Statements for Consecutive Datasets

Access Mode	Transmission Statements
SEQUENTIAL INPUT	READ FILE(<reference>)INTO(<reference>); READ FILE(<reference>)IGNORE(<expression>);
SEQUENTIAL OUTPUT	WRITE FILE(<reference>)FROM(<reference>);
SEQUENTIAL UPDATE	READ FILE(<reference>)INTO(<reference>); READ FILE(<reference>)IGNORE(<expression>); REWRITE FILE(<reference>)FROM(<reference>); DELETE FILE(<reference>);

An example of updating a direct file with the SEQUENTIAL and UPDATE attributes is:

```
OPEN FILE (FILE D) SEQUENTIAL UPDATE RECORD
  ENVIRONMENT (DIRECT, RECSIZE(N1));
```

```
  .
  .
  .
```

```
READ FILE (FILE D) INTO (VAR);
REWRITE FILE (FILE D) FROM (VAR1);
```

```
  .
  .
  .
```

```
READ FILE (FILE D) INTO (VAR);
DELETE FILE (FILE_D)
```

- 1) The first record in the file FILE_D is replaced by data in VAR1.
- 2) The second record in FILE_D is deleted.

An example of updating a direct file with the DIRECT and UPDATE attributes is:

```
OPEN FILE (FILE_D) DIRECT UPDATE RECORD
  ENVIRONMENT (DIRECT, RECSIZE(N1));
```

```
  .
  .
  .
```

```
READ FILE (FILE D) INTO (VAR) KEY(5);
DELETE FILE (FILE D) KEY(6);
WRITE FILE (FILE D) FROM (VAR1) KEYFROM(7);
REWRITE FILE (FILE D) FROM (VAR2) KEY(8);
```

Note: At the time the file FILE D is opened, records 5, 6, and 8 must exist. Record 7 must not exist.

- 1) Record number 5 is read
- 2) Record number 6 is deleted.
- 3) Record number 7 is added.
- 4) Record number 8 is replaced with a new record.

Example:

```
OPEN FILE (FILE_D) DIRECT OUTPUT RECORD  
  ENVIRONMENT (DIRECT, RECSIZE(N1), MAXKEY(N2));  
  .  
  .  
  .
```

```
WRITE FILE (FILE_D) FROM (VAR) KEYFROM (KX);
```

- 1) FILE_D is the file-name of the file which is created.
- 2) VAR is a variable in connected storage which is the source of the data being written.
- 3) N1 is an integer variable representing the record size.
- 4) N2 is an integer variable representing the maximum key value permitted.
- 5) KX is a variable representing the actual key value.

8.4.6.4 Updating a Direct Dataset

A direct dataset can be updated using the file attributes SEQUENTIAL and UPDATE with the READ and REWRITE statements or with the READ and DELETE statements. A direct file can also be updated using the DIRECT and UPDATE attributes with the REWRITE, DELETE, WRITE, or DELETE statements.

8.4.6.2 Updating a Consecutive Dataset

A dataset can be updated using the file attributes SEQUENTIAL and UPDATE with the READ, REWRITE, and DELETE statements. The <rewrite statement> updates the last record read. The <delete statement> deletes the last record read. The <rewrite statement> or <delete statement> must be preceded by a <read statement> without any intervening transmission statements for the file.

Example:

```
OPEN FILE (FILE_C) SEQUENTIAL UPDATE RECORD
  ENVIRONMENT (consecutive, RECSIZE(N2));
.
.
.
READ FILE (FILE_C) INTO (VAR);
.
.
.
REWRITE FILE (FILE_C) FROM (VAR1);
READ FILE (FILE_C) INTO (VAR);
DELETE FILE (FILE_C);
```

The first record in FILE_C is replaced by data in VAR1. The second record in FILE_C is deleted.

8.4.6.3 Creating a Direct Dataset

A direct dataset is created using the file attributes DIRECT and OUTPUT or DIRECT and UPDATE.

8.4.5.6 Keyfrom Option

The <keyfrom option> applies to a direct file organization with DIRECT and KEYED attributes. The option can be used in a <write statement> for a file with the OUTPUT attribute or a file with the DIRECT and UPDATE attributes.

Syntax:

```
<keyfrom option> ::= KEYFROM( <expression> )
```

The expression value specifies the location within a dataset where a record is to be written.

8.4.6 Data Transmission Examples

8.4.6.1 Creating a Consecutive Dataset

A consecutive dataset is created using the file attributes SEQUENTIAL and OUTPUT with the <write statement>.

Example:

```
OPEN FILE (FILE_C) SEQUENTIAL OUTPUT RECORD
  ENVIRONMENT (consecutive, RECSIZE(N2));
```

```
  .
  .
  .
```

```
WRITE FILE (FILE_C) FROM (VAR);
```

```
  .
  .
  .
```

```
CLOSE FILE (FILE_C) ENVIRONMENT (rewind);
```

- 1) FILE_C is the file name of the file which is created.
- 2) VAR is a variable in connected storage which is the source of the data being written.
- 3) N2 is an integer variable representing the record size.

Note: The key words in the ENVIRONMENT statement are for example only and may not represent those ultimately defined for the implementation.

SECTION 4 SYNTAX

Section 4.2

```

<reference> ::= <simple reference>
             | <subscripted reference>
             | <selected reference>
             | <qualified reference>
             | <function reference>

<simple reference> ::= <identifier>

<subscripted reference> ::=
    <array reference> [ <subscript> ]
    | <array reference>
      [ <subscript> { , <subscript> }... ]

<array reference> ::= <simple reference>

<subscript> ::= <expression> | *
              | <selector>

<selector> ::= & | & <selection qualifier>
              | <first element> : <last element>

<selection qualifier> ::= <reference>

<first element> ::= <expression>
<last element> ::= <expression>

<qualified reference> ::= <containing reference>
    [ . <containing reference> ]...
    . <member reference>
<containing reference> ::= <simple reference>
    | <subscripted reference>
<member reference> ::= <simple reference>
    | <subscripted reference>
    | <function reference>

<function reference> ::= <entry reference>
    [ <argument list> ]
<entry reference> ::= <reference>
<argument list> ::= ( [ <arguments> ] )
<arguments> ::= <expression> [ , <expression> ]...

```



```

<constant> ::= CONSTANT

<controlled> ::= CONTROLLED

<defined attribute> ::= DEFINED <base reference>
<base reference> ::= ( <base reference> )

<dimension> ::= [ DIMENSION ]
                ( <extent> [ ,<extent> [ ,<extent> ] ] )
<extent> ::= <expression> | *

<external> ::= EXTERNAL

<fixed> ::= FIXED

<float> ::= FLOAT

<initial> ::= INITIAL <initial list>
<initial list> ::= ( <initial item>
                    [ ,<initial item>... ] )
<initial item> ::= [<factor>]<initial value>
<factor> ::= ( <cvi expression> )
<initial value> ::= [+|-] <literal constant>

<length> ::= ( <string length> )
<string length> ::= <cvi expression> | *

<memory> ::= MEMORY( <memory type> )
<memory type> ::= MDA | MAIN

<precision> ::= ( <number of digits>
                  [ , <scale factor> ] )
<number of digits> ::= <cvi expression>
<scale factor> ::= [+|-] <cvi expression>

<range> ::= RANGE ( <low value>
                   : <high value> [ , <scale> ] )

<low value> ::= <cvi expression>
<high value> ::= <cvi expression>
<scale> ::= <cvi expression>

<static> ::= STATIC

<unaligned> ::= UNALIGNED

<varying> ::= VARYING

```

SECTION 3 SYNTAX

Section 3.5

```

<declare statement> ::= DECLARE <declaration component>
                        [ , <declaration component> ]...;
<declaration component> ::= [<level>] <declared name>
                        [<attribute set>]
                        | [ <level> ]
                        ( <factored name> [ , <factored name> ]... )
                        [<attribute set>]
<level> ::= <decimal integer>
<factored name> ::= <declared name> [ <dimension> ]
<declared name> ::= <identifier>
<attribute set> ::= <attribute> [ <attribute> ]...
<attribute> ::= <data type> | <storage class>
                | <alignment> | <initial>
                | <constant> | <dimension>
                | <external> | <defined attribute>
<data type> ::= <arithmetic> | <string>
                | <entry> | <file>
                | <builtin>
<arithmetic> ::= <float> <precision>
                | { <cardinal> | <fixed> }
                  { <precision> | <range> }
<string> ::= { <bit> | <character> }
              { <length> [ <varying> ]
                | <varying> <length> }
<storage class> ::= <automatic> | <controlled>
                  | <static>
<alignment> ::= <aligned> | <unaligned>
               | <memory>

```

Section 3.6

```

<aligned> ::= ALIGNED ( <boundary> )
<boundary> ::= BYTE | HALFWORD | WORD
              | <cvi expression>

<automatic> ::= AUTOMATIC

<bit> ::= BIT

<builtin> ::= BUILTIN

<cardinal> ::= CARDINAL

<character> ::= CHARACTER

```

```

{   B ' <bit chars>... '
    | Q ' <quad chars>... '
    | O ' <octal chars>... '
    | X ' <hex chars>... ' }

<bit chars> ::= 0 | 1 | <space>
<quad chars> ::= 2 | 3 | <bit chars>
<octal chars> ::= 4 | 5 | 6 | 7 | <quad chars>
<hex chars> ::= 8 | 9 | A | B | C | D | E | F
               | <octal chars>

<character-string constant> ::= [ (<cvi expression>) ]
                               ' [<character>]... '

<arithmetic constant> ::= <decimal constant>
                        | <binary fixed-point constant>
<binary fixed-point constant> ::=
    {0|1}... [ . [ {0|1}... ] ] B
    | . {0|1}... B
<decimal constant> ::= <decimal fixed-point constant>
                    | <floating-point constant>
<floating-point constant> ::= <mantissa><exponent>
<mantissa> ::= <decimal fixed-point constant>
<exponent> ::= E[+ | -]<decimal integer>

<decimal fixed-point constant> ::=
    <decimal integer>
    [ . [<decimal integer>] ]
    | . <decimal integer>
<decimal integer> ::= <digit>...

<i-sub> ::= <decimal integer> SUB

```

```

| <open statement>
| <read statement>
| <return statement>
| <revert statement>
| <rewrite statement>
| <signal statement>
| <write statement>

```

```

<prefix> ::= <label prefix> | <case prefix>
<label prefix> ::= <identifier>:
<case prefix> ::= CASE ( <case number> ) :
<case number> ::= <cvi expression>
                | <cvi expression> : <cvi expression>

```

Section 2.5

```

<token> ::= <identifier> | <literal constant>
          | <delimiter> | <i-sub>

<delimiter> ::= <graphic delimiter> | <space> | <comment>
<graphic delimiter> ::= + | - | * | / | ~ | &
                      | : | ; | ( | ) | _ | ||
                      | , | . | [ | ]
                      | = | ~= | <= | >=
                      | < | > | ~< | ~>

<space> ::= <blank> | <newline> | <newpage> | <tab>
<blank> ::= ASCII blank character
<newline> ::= ASCII new line character
<newpage> ::= ASCII new page character
<tab> ::= ASCII horizontal tab or ASCII vertical tab

<comment> ::= /* ASCII characters except an asterisk
              followed by a slash */

<identifier> ::= <letter>
                | <letter> | <digit> | _ | $ | ...

<letter> ::= A | B | C | D | E | F | G | H | I
           | J | K | L | M | N | O | P | Q | R
           | S | T | U | V | W | X | Y | Z

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<literal constant> ::= <bit-string constant>
                     | <character-string constant>
                     | <arithmetic constant>

<bit-string constant> ::= [ (<cvi expression>) ]

```


Appendix A

SECTION 2 SYNTAX

Section 2.1

```

<program> ::= <program unit>...
<program unit> ::= <external procedure>
<external procedure> ::= <procedure>

```

Section 2.2

```

<block> ::= <procedure>
<procedure> ::= <procedure statement>
               <procedure body>
               <endproc statement>
<procedure body> ::= <procedure component>
                    [ <routine statement>
                      <procedure component> ]...
<procedure component> ::= <statement>...
<statement list> ::= <statement>...
<statement> ::= <procedure>
               | <declare statement>
               | <basic statement>
               | <prefix><basic statement>
<basic statement> ::= <group>
                    | <independent statement>
<independent statement> ::= <single statement>
                           | <conditional statement>
                           | <on statement>

```

Section 2.3

```

<group> ::= <iterative loop>
          | <repetitive loop>
          | <case group>

```

Section 2.4

```

<single statement> ::= <allocate statement>
                     | <assignment statement>
                     | <call statement>
                     | <close statement>
                     | <delete statement>
                     | <free statement>
                     | <goto statement>
                     | <null statement>

```

Side-effect	4.2.10
SIGNAL	9.2.1
Signal	9.2
Signalled	5.6.2
SIN	4.7.3.3
SIZE	9.1.9
SNAP	9.2
SOME	4.7.1.11
SQRT	4.7.3.4
STATIC	3.6.20
STORAGE	9.1.10
STRING	4.7.1.12
STRINGRANGE	9.1.11
STRINGSIZE	9.1.12
Structure	3.3.3
SUBSCRIPTRANGE	9.1.13
SUBSTR builtin function	4.7.1.14
SUBSTR pseudo-variable	7.1.1.4
SUBTRACT	4.7.2.18
SUM	4.7.4.3
Syntactic-expression	1.4.3
Syntactic-unit	1.4.3
Syntax	1.4.2
SYSTEM	9.2
TAN	4.7.3.5
Terminal	1.4.2
THEN	6.4
TITLE	8.3.3
TO	6.1
TRANSMIT	9.1.14
TRUNC	4.7.2.19
UNALIGNED	3.6.21
Unconnected array	3.4.3
UNDEFINEDFILE	9.1.15
UNDERFLOW	9.1.16
UPDATE	8.3.1.5
Variable	3.1
VARCHAR	4.7.1.14
VARYING	3.6.22
VERIFY	4.7.1.15
WHILE	6.2
WORD	3.6.1
WRITE	8.4.2
ZERODIVIDE	9.1.17

Terms

MEMORY attribute	3.6.17
Meta-language	1.4
MIN	4.6.2.14
MOD	4.7.2.15
MULTIPLY	4.7.2.16
Named constant	3.6.6
Non-local GOTO	5.6
Non-terminal	1.4.2
ON	9.2
OPEN	8.3.3
Operand	4
Output	8.1
OVERFLOW	9.1.7
Parameters	4.2.9
Partially qualified	4.2.5
Passing arguments	4.2.8
PRECISION attribute	3.6.18
PRECISION builtin function	4.7.2.17
Precision rules	4.4.1.2
Prefix expression	4
Prefix operator	4.1.4
PROCEDURE	5.3.1
Procedure body	5.3
Production-rule	1.4.2
Program	2.1
Promotion	4.6
Qualified name	4.2.5
RANGE attribute	3.6.19
READ	8.4.1
RECORD	8.3.1.2
RECORD condition	9.1.8
Record oriented	8.1
REDUCIBLE	5.3.3.2
Reducible function	4.2.9
Reference	4.2
Resolved	4.2
RETURN	6.5
RETURNS	5.3.3.3
REVERT	9.2.2
REWRITE	8.4.3
ROUTINE	5.3.2
Row-major order	3.3.2
Scalar	3.3.1
Scope	3.5.1
SELECT	4.7.4.2
SEQUENTIAL	8.3.1.6
Sharing	3.3.6

FLOAT data type	3.6.14
Floating-point	3.2.3
FLOOR	4.7.2.8
Flow of control	5.1
FREE	7.3
FROM	8.4.5.3
Fully qualified	4.2.5
Generation of storage	4.2
GO	6.6
GOTO	6.6
HALFWORD	3.6.1
HIGH	4.7.1.7
IF	6.4.1
IFARRAY	6.4.2
IGNORE	8.4.5.4
Implementation defined	1.6
Immediately contained	2.2
INDEX	4.7.1.8
INDEX_FIRST	4.7.2.9
INDEX_MAX	4.7.2.10
INDEX_MIN	4.7.2.11
Infix expression	4
Infix operator	4.1.4
INITIAL	3.6.15
INPUT	8.1
Interpretation	1.4.4
INTERVAL_TEST	4.7.2.12
INTO	8.4.5.2
IRREDUCIBLE	5.3.3.1
Irreducible function	4.2.10
I-sub	3.3.6.3
KEY	8.4.5.5
KEY condition	9.1.6
KEYED	8.3.1.8
KEYFROM	8.4.5.6
LENGTH attribute	3.6.16
LENGTH builtin function	4.7.1.9
Level-one variable	3.3.3
Lexical syntax	2.5
Local GOTO	5.5
Logical IF	6.4.3
LOG2	4.7.3.2
LOW	4.7.1.10
MAIN	3.6.17
Main structure	3.5.2
MAX	4.7.2.13
MDA	3.6.17

Terms

COS	4.7.3.1
Cross-section	4.2.3
Dataset	8.1
Declare statement	3.5.2
Declaration	3.5
DECLARE	3.5.2
Defined	7.1
DEFINED attribute	3.6.9
Defining production-rule	1.4.2
DELETE	8.4.4
DIM	4.7.4.1
DIMENSION	3.6.10
Direct	8.2.3
DIVIDE	4.7.2.5
DO	6.1
Do while loop	6.2
Do loop	6.1
Dynamic descendent	5.4.1
Dynamic predecessor	5.4.1
ELSE	6.4
ELSECASE	6.3
ENDCASE	6.3
ENDDO	6.1
ENDFILE	9.1.2
ENDIF	6.4
ENDON	9.2
ENDPROC	5.3.1
Entry activation	5.3.3
Entry attribute	3.6.11
ENVIRONMENT	8.3.1.9
Environment pointer	5.4.2
ERROR	9.1.3
Established	5.6.2
Evaluated	4.2
EVERY	4.7.1.6
Extent	3.2
EXTERNAL	3.6.12
External procedure	2.1
FILE	8.3
File attributes	8.3.1
Filename	8.3
FINISH	9.1.4
FIXED builtin function	4.7.2.6
FIXED data type	3.6.13
Fixed-point	3.2.2
Fixed-point integer	3.2.2
FIXEDOVERFLOW	9.1.5
FLCAT builtin function	4.7.2.7

Terms

ABS	4.7.2.1
Activated	5.4.1
Activation record	5.4.1
ADD	4.7.2.2
Aggregate promotion	4.6
Aggregate value	4.2.1
ALIGNED	3.6.1
ALLOCATE	7.2
ALLOCATION	4.7.5.1
Applicable	4.2.6
Argument	4.2.9
Array	3.3.2
Array of structures	3.3.4
AUTOMATIC	3.6.2
Basic expression	4
BIT	3.6.3
Bit-string	3.2.4
Block activation	5.4.1
Block structure	2.1
BOOL	4.7.1.2
Builtin attribute	3.6.4
Built-in function	4.7
BY	6.1
By-reference	4.2.8
By-value	4.2.8
BYTE	3.6.1
CALL	6.5
CARDINAL attribute	3.6.5
CASE	6.3
Category name	1.4.1
CEIL	4.7.2.4
CHARACTER	3.6.6
Character-string	3.2.5
CLOSE	8.3.5
COLLATE	4.7.1.4
Condition	9.1
Conditional Statement	6.4
Conformance	4.6
Connected Array	3.4.3
Consecutive	8.2.2
Constant	2.5.2, 3.1
CONSTANT attribute	3.6.7
Contained	2.2
CONTROLLED	3.6.8
CONVERSION	9.1.1
Conversion, implicit	4.5
Conversion functions	4.7
COPY	4.7.1.5

```

<file option> ::= FILE( <reference> )
<into option> ::= INTO( <reference> )
<from option> ::= FROM( <reference> )
<ignore option> ::= IGNORE( <expression> )
<key option> ::= KEY( <expression> )
<keyfrom option> ::= KEYFROM( <expression> )

```

SECTION 9 SYNTAX

Section 9.1

```

<condition name> ::= CONVERSION
| ENDFILE ( <reference> )
| ERROR | FINISH
| FIXEDOVERFLOW
| KEY ( <reference> )
| OVERFLOW
| RECORD ( <reference> )
| SIZE | STORAGE | STRINGRANGE
| STRINGSIZE | SUBSCRIPTRANGE
| TRANSMIT ( <reference> )
| UNDEFINEDFILE ( <reference> )
| UNDERFLOW | ZERODIVIDE

```

Section 9.2

```

<on statement> ::= ON <condition name> <on-unit>
<on-unit> ::= <statement list> ENDON;
| [SYSTEM] ;

<revert statement> ::= REVERT <condition name>
| [ ,<condition name>]... ;

<signal statement> ::= SIGNAL <condition name>;

```

SECTION 8 SYNTAX

Section 8.3

```

<file attribute> ::= FILE

<record attribute> ::= RECORD

<input attribute> ::= INPUT
<output attribute> ::= OUTPUT

<update attribute> ::= UPDATE
<sequential attribute> ::= SEQUENTIAL
<direct attribute> ::= DIRECT
<keyed attribute> ::= KEYED

<environment> ::= ENVIRONMENT( <option list> )

<open statement> ::= OPEN <single open>
                    [, <single open> ] ;

<single open> ::= <file option>
                 { [ RECORD ]
                   * [ <title option> ]
                   * { INPUT | OUTPUT | UPDATE }
                   * { SEQUENTIAL | DIRECT [ KEYED ] }
                   * [ <environment> ] }

<title option> ::= TITLE( <expression> )

<close statement> ::= CLOSE <single close>
                    [, <single close> ] ;
<single close> ::= <file option>
                  <environment>

```

Section 8.4

```

<read statement> ::= READ <file option>
                  { <into option> | <ignore option> }
                  [ <key option> ] ;

<write statement> ::= WRITE <file option> <from option>
                   [ <keyfrom option> ] ;

<rewrite statement> ::= REWRITE <file option> <from option>
                       [ <key option> ] ;

<delete statement> ::= DELETE <file option>
                       [ <key option> ] ;

```


<logical if> ::= IF (<expression>)

Section 6.5

<call statement> ::= CALL <entry reference>
[<argument list>] ;

<return statement> ::= RETURN [(<function value>)] ;
<function value> ::= <expression>

Section 6.6

<goto statement> ::= GO TO <identifier> ;
| GOTO <identifier> ;

Section 6.7

<exit statement> ::= EXIT [<identifier>] ;

SECTION 7 SYNTAX

Section 7.1

<assignment statement> ::= <target> = <expression> ;
<target> ::= <reference> | <pseudo-variable>
<pseudo-variable> ::= <substr pseudo>
<substr pseudo> ::= substr (<reference>
 , <expression> [, <expression>])

Section 7.2

<allocate statement> ::= ALLOCATE <allocate list> ;
<allocate list> ::= <allocate reference>
 [, <allocate reference>] ...
<allocate reference> ::= <identifier>

Section 7.3

<free statement> ::= FREE <reference>
 [, <reference>] ... ;

SECTION 6 SYNTAX

Section 6.1

```

<iterative loop> ::= DO <step definition> <group body>
                      ENDDO [ <identifier> ] ;
<step definition> ::= <index> = <expression>
                      <iteration control> ;
<index> ::= <identifier>
<iteration control> ::= TO <limit> [ BY <increment> ]
<limit> ::= <expression>
<increment> ::= <expression>
<group body> ::= <basic statement>...

```

Section 6.2

```

<repetitive loop> ::= <do while> <group body>
                      ENDDO [ <identifier> ] ;
<do while> ::= DO WHILE ( <expression> ) ;
<group body> ::= { [ <label prefix> ]
                  <basic statement> }...

```

Section 6.3

```

<case group> ::= <case heading> <case body> <endcase>
<case heading> ::= DO CASE <expression> ;
<case body> ::= <case section>... [ <else case> ]
<case section> ::= <case prefix>... <basic statement>...
<case prefix> ::= CASE ( <cvi expression>
                       [ : <cvi expression> ] ) :
<else case> ::= ELSECASE <basic statement>...
<endcase> ::= ENDCASE [ <identifier> ] ;

```

Section 6.4

```

<conditional statement> ::= <if statement>
                          | <ifarray statement>
                          | <logical if statement>
<if statement> ::= <if clause> <if body> <endif>
<ifarray statement> ::= <ifarray clause>
                      <if body> <endif>
<if clause> ::= IF <expression>
<ifarray clause> ::= IFARRAY <expression>
<if body> ::= <true part> [ <else part> ]
<true part> ::= THEN <basic statement>...
<else part> ::= ELSE <basic statement>...
<endif> ::= ENDIF [ <identifier> ] ;
<logical if statement> ::= <logical if>
                        <single statement>

```

```

<procedure body> ::= <procedure component>
                    [ <routine statement>
                      <procedure component> ]...

<procedure component> ::= <statement> ...

<endproc statement> ::= ENDPROC <closure identifier> ;
<closure identifier> ::= <identifier>

<procedure statement> ::= <label prefix> PROCEDURE
                          [ ( [ <parameter list> ] ) ]
                          [ <procedure option> ]... ;

<procedure option> ::= <entry option> | RECURSIVE

<entry option> ::= <reducible option>
                  | <returns attribute>

<parameter list> ::= <identifier>
                    [ , <identifier> ] ...

<reducible option> ::= <irreducible attribute>
                      | <reducible attribute>
<irreducible attribute> ::= IRREDUCIBLE
<reducible attribute>  ::= REDUCIBLE

<returns attribute> ::= RETURNS
                     ( <returns descriptor> )
<returns descriptor> ::= <descriptor>
                       [ , <descriptor> ]...
<descriptor> ::= <level> [ <attribute set> ]
                 | [ <level> ] <attribute set>

<routine statement> ::= <label prefix> ROUTINE
                       [ ( [ <parameter list> ] ) ]
                       [ <entry option> ] ;

<entry> ::= ENTRY [ (
                  [ <parameter descriptor list> ] ) ]
                  [ <entry option> ]

<parameter descriptor list> ::= <parameter descriptor>
                               [ , <parameter descriptor> ]...

<parameter descriptor> ::= <descriptor>

```

Section 4.3

```

<expression> ::= <expression one>
               | <expression> | <expression one>

<expression one> ::= <expression two>
                   | <expression one> & <expression two>

<expression two> ::= <expression three>
                   | <expression three> <relational> <expression three>

<relational> ::= = | ~= | ≤ | ~≤ | ≤= | ≥ | ~≥ | ≥=

<expression three> ::= <expression four>
                    | <expression three> || <expression four>

<expression four> ::= <expression five>
                   | <expression four> { + | - } <expression five>

<expression five> ::= <expression six>
                   | <expression five> { * | / } <expression six>

<expression six> ::= <basic expression>
                  | <simple expression>
                  | <parenthesized expression> | <expression seven>

<expression seven> ::= { <basic expression>
                       | <parenthesized expression> }
                     ** <expression six>

<simple expression> ::= { + | - | ~ } <expression six>

<parenthesized expression> ::= ( <expression> )

<basic expression> ::= <reference>
                    | <literal constant> | <i-sub>

<cvi expression> ::= <expression>

```

SECTION 5 SYNTAX

Section 5.3

```

<procedure> ::= <procedure statement>
              <procedure body> <endproc statement>

```


Sample Program Report

```
/* Store total number of points in current  
processed line */  
  
vector_number_points[index] = number_of_points ;  
ENDDO ;  
ENDPROC follow_lines ;
```

```

DO dir = prev_direction TO
    prev_direction + number_of_directions ;
cur_dir = MOD( dir-1,8 ) ;

/* Check direction cur_dir for a one bit */

DO CASE cur_dir ;

CASE(0):      IF(data[i,j+1])..
               found_succeeding_point = B'1' ;
CASE(1):      IF(data[i+1,j+1])
               found_succeeding_point = B'1' ;
CASE(2):      IF(data[i+1,j])
               found_succeeding_point = B'1' ;
CASE(3):      IF(data[i+1,j-1])
               found_succeeding_point = B'1' ;
CASE(4):      IF(data[i,j-1])
               found_succeeding_point = B'1' ;
CASE(5):      IF(data[i-1,j-1])
               found_succeeding_point = B'1' ;
CASE(6):      IF(data[i-1,j])
               found_succeeding_point = B'1' ;
CASE(7):      IF(data[i-1,j+1])
               found_succeeding_point = B'1' ;
ENDCASE ;
IF( found_succeeding_point ) EXIT;
ENDDO ;

IF found_succeeding_point THEN

    /* Enter direction of discovered point */

    vector_list[index,number_of_points] = cur_dir ;
    number_of_points = number_of_points + 1 ;

    /* Start search for next point based on
    direction of current point */

    prev_direction = next_direction[cur_dir+1] ;

    /* Number of possible directions to try is based
    on direction of current point */

    number_of_directions = table_of_tries[cur_dir+1] ;
ENDIF ;
ENDDO ;

```

```

/* Clear boarder around data area */

data[1,*] = 0;
data[194,*] = 0;
data[* ,1] = 0;
data[* ,194] = 0;

/* Examine bit vector start.tag for ones, continue
to loop until all start points have been processed
*/

DO WHILE( SOME(start.tag) ) ;

    /* Get index of first element set to one */

    index = INDEX_FIRST( start.tag ) ;

    /* Eliminate from further processing */

    start[index].tag = B'0' ;

    /* Start looking in direction 1 and do up to
    7 additional directions as necessary */

    prev_direction = 1 ;

    /* Try next 7 possible directions if #1 fails */

    number_of_directions = 7 ;

    /* Obtain coordinates of start point */

    i = start[index].x ;
    j = start[index].y ;

    /* Start with state of "next point in the line
    has been found" */

    found_succeeding_point = B'1' ;

    number_of_points = 1 ;

    DO WHILE( found_succeeding_point ) ;

        /* Assume end point already found */

        found_succeeding_point = B'0' ;

    line_follow:

```

of one means another point was found, a value of zero means the end of a line has been detected.

NUMBER_OF_POINTS is a variable containing the number of points (or vectors) discovered in a line.

DIR is a do-loop index and represents the range of directions to be tried.

CUR_DIR is the direction currently being tried and is a value between 0 and 7, reference Figure 1.

VECTOR_LIST is a two dimensional array containing the results of the line following procedure. It has a capacity for 32 individual lines, each containing up to 256 vector values (up to 256 points).

VECTOR_NUMBER_POINTS is an array containing the number of points for each of the 32 possible lines contained in VECTOR_LIST.

2.2 LINE FOLLOWING PROCEDURE

```
follow_lines: PROCEDURE;
  DECLARE data(194,194) BIT(1) MEMORY(mda),
         prev_direction CARDINAL(4),
         number_of_directions CARDINAL(4);

  DECLARE index CARDINAL(5),
         1 start(32) MEMORY(mda),
         2 tag BIT(1),
         2 x CARDINAL(16),
         2 y CARDINAL(16);

  DECLARE found_succeeding_point BIT(1),
         number_of_points CARDINAL(16),
         (dir,cur_dir) CARDINAL(4),
         (i,j,index) CARDINAL(8),
         vector_list(32,256) CARDINAL(4),
         table_of_tries(8) CARDINAL(4) CONSTANT
             INITIAL(3,5,3,5,3,5,3,5),
         next_direction(8) CARDINAL(4) CONSTANT
             INITIAL(7,7,1,1,3,3,5,5),
         vector_number_points(32) CARDINAL(16);
```


parallel) such functions as line thinning (reducing a line to a single cell in width), clutter elimination, and tagging all lines in the array by storing their starting coordinates in a table. A starting coordinate may be a boundary point or an interior point. Beginning with a starting point, the procedure follows the points along a line segment vectorizing it as it proceeds.

2.1 Definition of Variable Names

DATA is a two dimensional array of single bit items and contains the lines to be vectorized. It is oversized by one cell on each border so that all starting points will be "interior points". In this way no check need be made when following a line to insure the dimensions of DATA are not exceeded.

START.TAG is an array of single bit items containing a one bit value in each index position of a starting coordinate, i.e., values in the arrays START.X and START.Y are selected.

START.X is an array of x coordinate starting point values of the various lines contained in the DATA array.

START.Y is an array of y coordinate starting point values of the various lines contained in the DATA array.

PREV_DIRECTION is a variable containing the direction of the previous successful search along the line. In Figure 1 the column of values under "LATEST POINT" correspond with PREV_DIRECTION values.

NEXT_DIRECTION is an array of values containing the first direction of search for the point along the line, and is a function of PREV_DIRECTION. In Figure 1 the column of values under "FIRST DIRECTION FOR NEXT CHECKED POINT" correspond with NEXT_DIRECTION values.

TABLE_OF_TRIES is an array containing the maximum number of possible directions in which the succeeding point along a line may be found. In Figure 1 the column of values under "# DIRECTIONS TO CHECK" correspond with TABLE_OF_TRIES values.

FOUND_SUCCEEDING_POINT is a single bit tag indicating whether the line following procedure was successful at detecting a succeeding point. A value

1. INTRODUCTION

Two applications were selected to illustrate the facilities of the STAPAN high order language defined in the specification document.

The applications chosen demonstrate the language facilities in general, and the special features of the language that make use of the STARAN's unique architecture.

The first sample program, line following, shows the advantage of STARAN in an application that requires logic and bit manipulations on single bit items. The language, unlike FORTRAN, provides the required facilities to operate on bit vectors. This application is not practical on a sequential machine programmed in FORTRAN.

The second sample program, image magnification, demonstrates the advantage of STARAN in performing parallel arithmetic operations. The language provides parallel vector operations on variable precision numbers. These facilities are not available or practical in a high level language on a sequential machine. The cubic convolution routine which is a part of this program illustrates three methods of programming to show various levels of control provided in the language.

In both sample programs, a comparison was made to a version written in APPLE, the STARAN assembly language. The ratio of high level language instructions (statements) to APPLE instructions was approximately 1 to 5.

Since both applications have unique requirements that are not easily expressed in existing high level languages, a meaningful comparison to one of these languages cannot be made.

2. LINE FOLLOWING

One of the reasons for choosing this algorithm as a candidate for coding is the nature of the process performed. The data being processed is essentially binary ones and zeroes representing the raster scanned data of a map. It is observed that the degree of effort and clarity of representation of this algorithm as written in high level language are good tests for the programming language.

The line following procedure vectorizes all lines in an array and outputs the data to various tables. A previous routine has performed (in

Sample Program Report

<div>567</div> <div>4X0</div> <div>321</div>	EIGHT DIRECTIONAL VALUES	LATEST POINT	FIRST DIRECTION FOR NEXT CHECKED POINT	# DIRECTIONS TO CHECK
<div> <div></div> <div></div> <div>*</div> </div> <div> <div>0</div> <div>N</div> <div>*</div> </div> <div> <div></div> <div></div> <div>*</div> </div>		0	7	3
<div> <div>0</div> <div></div> <div>*</div> </div> <div> <div></div> <div>N</div> <div>*</div> </div> <div> <div>*</div> <div>*</div> <div>*</div> </div>		1	7	5
<div> <div>0</div> <div></div> <div></div> </div> <div> <div></div> <div>N</div> <div></div> </div> <div> <div>*</div> <div>*</div> <div>*</div> </div>		2	1	3
<div> <div>*</div> <div></div> <div>0</div> </div> <div> <div>*</div> <div>N</div> <div></div> </div> <div> <div>*</div> <div>*</div> <div>*</div> </div>		3	1	5
<div> <div>*</div> <div></div> <div></div> </div> <div> <div>*</div> <div>N</div> <div>0</div> </div> <div> <div>*</div> <div></div> <div></div> </div>		4	3	3
<div> <div>*</div> <div>*</div> <div>*</div> </div> <div> <div>*</div> <div>N</div> <div></div> </div> <div> <div>*</div> <div></div> <div>0</div> </div>		5	3	5
<div> <div>*</div> <div>*</div> <div>*</div> </div> <div> <div></div> <div>N</div> <div></div> </div> <div> <div></div> <div>0</div> <div></div> </div>		6	5	3
<div> <div>*</div> <div>*</div> <div>*</div> </div> <div> <div></div> <div>N</div> <div>*</div> </div> <div> <div>0</div> <div></div> <div>*</div> </div>		7	5	5

Figure 1. Vector Increment Search Criteria
 N = new point, O = old point
 * = cells to check

3. IMAGE MAGNIFICATION USING CUBIC CONVOLUTION

This program reads pixel data from magnetic tape and writes the magnified pixel data on magnetic tape. The program uses the cubic convolution method for interpolating intermediate pixel values.

The program assumes a four to one magnification mainly for ease of reading and understanding. It should be noted that the magnification factor and other associated constants in the program could easily be defined as variables. The magnification is performed in one direction only.

The program consists of a main program which performs magnetic tape I/O, and two subroutines: DATAMAP and CUBIC.

DATAMAP arranges pixel data in the MDA memory to permit the parallel computations required in the CUBIC subroutine.

CUBIC performs the cubic convolution algorithm, and returns the magnified pixel values. The CUBIC subroutine is written using three different programming approaches to illustrate the programmer's flexibility in controlling the problem solution.

3.1 Definition of Variable Names

FILE_P is the magnetic tape input file; it contains the pixel data. The record lengths are 512 bytes which represent one scan line of 8-bit pixel data.

FILE_M is the output magnetic tape file; record lengths are 512 bytes representing one-fourth of a scan line of magnified pixel data. Four output records are generated for each input record.

N_PIXELS is the number of pixels in a scan line. It is set to 512 in this example.

PIX is a vector of 8-bit cardinal values. It contains the input pixel values for a scan line.

Sample Program Report

XU is a 4 element vector of 4-bit fractions. It represents the position of the interpolated (magnified) pixel values relative to the n-th input pixel value.

U is a vector of 4-bit fractions. It is the XU vector above replicated $N_PIXELS/4$ times(128).

PIXN0 is a vector of 8-bit cardinal values $N_pixels(512)$ in length. It contains the 128 pixel values from PIX; each pixel value is replicated 4 times to generate a vector of length 512.

PIXN contains the same values as PIXN0 shifted up four element positions with an additional pixel value appended to the end. This additional pixel value is also replicated 4 times.

PIXN1 is defined the same as PIXN shifted up 4 elements.

PIXN2 is defined the same as PIXN1 shifted up 4 elements.

*****NOTE: An MDA memory map is provided in the DATAMAP description which follows*****

READING is a single bit which is used as a flag to indicate an end-of-file condition on the input tape.

SEGNO is the number of the segment of the input vector(PIX) which is currently being processed. SEGNO = 1 for pixel values 1 through 128; 2 for pixel values 129 through 256; 3 for pixel values 257 through 384; 4 for pixel values 385 through 512.

B0 through B8 are intermediate values in methods 2 and 3 of the CUBIC procedure. They permit variable precision of intermediate computations and control the order of these computations.

3.2 Main Program

/*

MAIN_PROG functions:

- 1- Reads magnetic tape records representing a horizontal scan line of image data. Each record contains 512 (8-bit) pixel values.
- 2- Calls DATAMAP to arrange data in the MDA memory for parallel execution.
- 3- Calls CUBIC to perform the magnification function (CUBIC returns a vector of 512 8-bit pixel values)
- 4- Writes the magnified image to a magnetic tape(512 pixels per record). */

magnify4: PROCEDURE;

/*DEFINE INPUT AND OUTPUT FILE*/

```

DECLARE file_p FILE SEQUENTIAL INPUT RECORD,
        file_m FILE SEQUENTIAL OUTPUT RECORD;
DECLARE reading BIT(1),
        n_pixels CARDINAL(16) CONSTANT INITIAL(512),
        segno CARDINAL (4);

```

/*DEFINE PIXEL VECTORS*/

```

DECLARE (pixn0, pixn, pixn1, pixn2)(n_pixels)
        CARDINAL(8) MEMORY(mda) , /* positive values 0 to 255 */
        u(n_pixels) FIXED(4,4) MEMORY(mda),
        xu(4) FIXED(4,4),
        pix(n_pixels) CARDINAL(8);

```

/* Set increment values for 4 to 1 magnification ratio.*/

```

xu(1) = .00;
xu(2) = .25;
xu(3) = .50;
xu(4) = .75;
reading = b'1';

```

Sample Program Report

```

/*OPEN MAGNETIC TAPE FILES*/

OPEN FILE(file_p) ENVIRONMENT(CONSECUTIVE,RECSIZE(n_pixels)),
      (file_m) ENVIRONMENT(CONSECUTIVE,RECSIZE(n_pixels));

/* Read n_pixels pixel values into PIX */

READ FILE(file_p) INTO (pix);
ON ENDFILE (file_p) reading = b'0';
ENDON;
DO WHILE (reading);

/* Initialize counter to process n_pixels/4 of
   the input data. */

      segno = 1;

/* Process n_pixels/4 generating n_pixels of
   magnified pixel values.
   Generate 4 output records for each input record. */

      DO WHILE (segno < 5);
        CALL datamap (xu, segno);
        CALL cubic ;
        WRITE FILE (file_m) FROM (mpixn);
        segno = segno +1;
      ENDDO;
      READ FILE (file_p) INTO (pix);
    ENDDO;
CLOSE FILE (file_p), FILE (file_m);
END magnify4;

```

3.3 Data Mapping Routine

```

/*
  DATAMAP functions:
  1- Replicates pixel values four times each into
     vector PIXNO (512 values--128 pixels input
     times 4 replications)

  2- Copy PIXNO to PIXN shifting vector PIXNO up
     by four elements (i.e. PIXN(1) = PIXNO(5),
     PIXN(2) = PIXNO(6), etc)

  3- Copy PIXN to PIXN1 shifting vector PIXN up
     by four elements (i.e. PIXN1(1) = PIXN(5),
     PIXN1(2) = PIXN(6), etc)

```

Sample Program Report

- 4- Copy PIXN1 to PIXN2 shifting PIXN1 up by four elements (i.e. PIXN2(1) = PIXN1(5), PIXN2(2) = PIXN1(6),etc)
- 5- Replicate interpolation increment values through a vector of length 512 (i.e.U(1) = XU(1), U(2) = XU(2), U(3) = XU(3), U(4) = XU(4), U(5) = XU(1), U(6) = XU(2), ...U(511) = XU(3), U(512) = XU(4))

MDA Data Map

MDA WORD	PIXNO	PIXN	PIXN1	PIXN2	U	INTERMEDIATE VALUES	MPIXN
0	p(n-1)	p(n)	p(n+1)	p(n+2)	xu1	— — —	m(1)
1	p(n-1)	p(n)	p(n+1)	p(n+2)	xu2	— — —	m(2)
2	p(n-1)	p(n)	p(n+1)	p(n+2)	xu3	— — —	m(3)
3	p(n-1)	p(n)	p(n+1)	p(n+2)	xu4	— — —	m(4)
4	p(n)	p(n+1)	p(n+2)	p(n+3)	xu1	— — —	m(5)
5	p(n)	p(n+1)	p(n+2)	p(n+3)	xu2	— — —	m(6)
6	p(n)	p(n+1)	p(n+2)	p(n+3)	xu3	— — —	m(7)
7	p(n)	p(n+1)	p(n+2)	p(n+3)	xu4	— — —	m(8)
8	p(n+1)	p(n+2)	p(n+3)	p(n+4)	xu1	— — —	m(9)
9	p(n+1)	p(n+2)	p(n+3)	p(n+4)	xu2	— — —	m(10)
10	p(n+1)	p(n+2)	p(n+3)	p(n+4)	xu3	— — —	m(11)
11	p(n+1)	p(n+2)	p(n+3)	p(n+4)	xu4	— — —	m(12)
.							
.							
.							
508	p(128)	p(129)	p(130)	p(131)	xu1	— — —	m(509)
509	p(128)	p(129)	p(130)	p(131)	xu2	— — —	m(510)
510	p(128)	p(129)	p(130)	p(131)	xu3	— — —	m(511)
511	p(128)	p(129)	p(130)	p(131)	xu4	— — —	m(512)

*/

```
datamap: PROCEDURE(xu, segno);
```

```
/* Replicate n_pixels/4 pixel values four times each
to generate the vector pixn0. Also replicate the
incrementation values through n_pixels element vector.
```

*/

```
DECLARE (i, segno) CARDINAL(16);
DECLARE last_4 RANGE(n_pixels-3 : n_pixels);
i = 0;
DO WHILE (i <= n_pixels);
```


Sample Program Report

```

        u[i+1] =      xu[MOD(i,4)+1];
        pixn0[i+1] = pix[(FLOOR(i/4) +1)*segno];
        i = i + 1;
    ENDDO;

/* Generate vectors pixn, pixn1, pixn2, moving each up four
   elements and append next pixel value on the end
   replicated four times.
*/

    pixn[*] = pixn0[5:DIM(pixn0)];
    pixn[last_4] = pix[FLOOR(i/4)*segno+1];

    pixn1[*] = pixn[5:DIM(pixn)];
    pixn1[last_4] = PIX[FLOOR(i/4)*segno+2];

    pixn2[*] = pixn1[5:DIM(pixn1)];
    pixn2[last_4] = PIX[FLOOR(i/4)*segno+3];
    RETURN;
ENDPROC datamap;

```

3.4 Cubic Routines

```

/*
    CUBIC functions:

    1- Uses vector values produced in DATAMAP.
        (PIXNO, PIXN, PIXN1, PIXN2, U)

    2- Performs cubic algorithm to generate
        intermediate pixel values for the magnified
        image (Three programming methods are illustrated.)

    3- Returns final magnified pixel vector (512 pixels)
        to MAINPROC for output.(MPIXN)

```

Formula for cubic convolution:

$$\begin{aligned}
 r(n) = & ((p(n)-p(n-1))+p(n+2)-p(n+1)) * (u(n)-1) \\
 & -p(n)+p(n-1)) * u(n) + p(n+1)-p(n-1)) * u(n) \\
 & +p(n)
 \end{aligned}$$

In program:

```

    MPIXN = m(n)
    PIXNO = p(n-1)
    PIXN  = p(n)
    PIXN1 = p(n+1)

```

Sample Program Report

```
PIXN2 = p(n+2)
U      = u(n)
where n ranges from 1 to 512
*/
```

3.4.1 Cubic Method Number 1

```
/*METHOD 1*/
```

```
/* Compiler controls intermediate storage and order of
   computation(where not restricted by parentheses).
   Accuracy of intermediate values is language
   defined and controlled by the compiler.
*/
```

```
cubic: PROCEDURE;
  mpixn=(( ( pixn2 - pixn1 + pixn - pixn0)*(u - 1) - pixn
    + pixn0 ) *u + pixn1 - pixn0 ) *u + pixn;
  RETURN;
ENDPROC cubic;
```

3.4.2 Cubic Method Number 2

```
/*METHOD 2*/
```

```
/* Program controls intermediate storage and order of computation */
```

```
cubic: PROCEDURE;
DECLARE (b0, b1, b6)(n_pixels) FIXED(8) MEMORY(mda),
        b2(n_pixels) FIXED(9) MEMORY(mda),
        (b3, b4)(n_pixels) FIXED(14, 4) MEMORY(mda),
        (b5, b8)(n_pixels) FIXED(19, 8) MEMORY(mda),
        b7(n_pixels) FIXED(13, 4) MEMORY(mda);
```

```
b0 = pixn2 - pixn1;
b1 = pixn - pixn0;
b2 = b0 + b1;
b3 = b2 *(u - 1);
b4 = b3 - b1;
b5 = b4 * u;
b6 = pixn1 - pixn0;
b7 = b5 + b6;
b8 = b7 * u;
```

Sample Program Report

```
    mpixn = b8 + pixn;  
    RETURN;  
ENDPROC cubic;
```

3.4.3 Cubic Method Number 3

```
/*METHOD 3*/
```

```
/* Program controls intermediate storage, order of  
   computation and mda memory allocation.  
   The effect of this is very similar to method 1  
   as done by the compiler. */
```

```
cubic: PROCEDURE;  
DECLARE (b0, b1, b6)(n_pixels) FIXED(8) CONTROLLED MEMORY(mda),  
        b2(n_pixels) FIXED(9) CONTROLLED MEMORY(mda),  
        (b3, b4)(n_pixels) FIXED(14, 4) CONTROLLED MEMORY(mda),  
        (b5, b8)(n_pixels) FIXED(19, 8) CONTROLLED MEMORY(mda),  
        b7(n_pixels) FIXED(13, 4) CONTROLLED MEMORY (mda);  
ALLOCATE b0;  
    b0 = pixn2 - pixn1;  
ALLOCATE b1, b2;  
    b1 = pixn - pixn0;  
    b2 = b0 + b1;  
FREE b0;  
ALLOCATE b3;  
    b3 = b2 *(u - 1);  
FREE b2;  
    b4 = b3 - b1;  
FREE b1, b3;  
ALLOCATE b5;  
    b5 = b4 * u;  
FREE b4;  
ALLOCATE b6, b7;  
    b6 = pixn1 - pixn0;  
    b7 = b5 + b6;  
FREE b6, b5;  
ALLOCATE b8;  
    b8 = b7 * u;  
FREE b7;  
    mpixn = b8 + pixn;  
FREE b8;  
RETURN;  
ENDPROC cubic;
```

User Library Sample Procedure

A Data Formatting Procedure

```
list_array: PROCEDURE(x, wide);

/* List a two-dimensional float array on the standard */
/* output print file (list_output), "wide" items per */
/* line (or less if they will not fit) */

DECLARE x(*, *) FLOAT(*), wide FIXED(15);

/* buffer for one line of output */

DECLARE line_size FIXED(15) CONSTANT INITIAL(128);
DECLARE s1 CHARACTER(line_size) VARYING;
DECLARE (i, j, width) FIXED(15);

/*characters to display one "FLOAT" item */

width = VCHAR(x[1, 1]) +1;

DO i = 1 TO HBOUND(x, 1);
  s1 = 'ROW ' || VCHAR(i);
  k = 0;
  DO j = 1 TO HBOUND(x, 2);
    IF LENGTH(s1) + width > line_size | k >= wide
      THEN
        WRITE FILE(list_output) FROM(s1);
        s1 = ' ' ;
        k = 0;
      ENDIF;
    s1 = s1 || VCHAR(x[i, j]) || ' ' ;
    k = k+1;
  ENDDO;
  IF LENGTH(s1) > 4 THEN
    WRITE FILE(list_output) FROM(s1);
  ENDDO;
ENDPROC list_array;
```


METRIC SYSTEM

BASE UNITS:

Quantity	Unit	SI Symbol	Formula
length	metre	m	...
mass	kilogram	kg	...
time	second	s	...
electric current	ampere	A	...
thermodynamic temperature	kelvin	K	...
amount of substance	mole	mol	...
luminous intensity	candela	cd	...

SUPPLEMENTARY UNITS:

plane angle	radian	rad	...
solid angle	steradian	sr	...

DERIVED UNITS:

Acceleration	metre per second squared	...	m/s
activity (of a radioactive source)	disintegration per second	...	(disintegration)/s
angular acceleration	radian per second squared	...	rad/s
angular velocity	radian per second	...	rad/s
area	square metre	...	m
density	kilogram per cubic metre	...	kg/m
electric capacitance	farad	F	A·s/V
electrical conductance	siemens	S	A/V
electric field strength	volt per metre	...	V/m
electric inductance	henry	H	V·s/A
electric potential difference	volt	V	W/A
electric resistance	ohm	...	V/A
electromotive force	volt	V	W/A
energy	joule	J	N·m
entropy	joule per kelvin	...	J/K
force	newton	N	kg·m/s
frequency	hertz	Hz	(cycle)/s
illuminance	lux	lx	lm/m
luminance	candela per square metre	...	cd/m
luminous flux	lumen	lm	cd·sr
magnetic field strength	ampere per metre	...	A/m
magnetic flux	weber	Wb	V·s
magnetic flux density	tesla	T	Wb/m
magnetomotive force	ampere	A	...
power	watt	W	J/s
pressure	pascal	Pa	N/m
quantity of electricity	coulomb	C	A·s
quantity of heat	joule	J	N·m
radiant intensity	watt per steradian	...	W/sr
specific heat	joule per kilogram-kelvin	...	J/kg·K
stress	pascal	Pa	N/m
thermal conductivity	watt per metre-kelvin	...	W/m·K
velocity	metre per second	...	m/s
viscosity, dynamic	pascal-second	...	Pa·s
viscosity, kinematic	square metre per second	...	m/s
voltage	volt	V	W/A
volume	cubic metre	...	m
wavenumber	reciprocal metre	...	(wave)/m
work	joule	J	N·m

SI PREFIXES:

Multiplication Factors	Prefix	SI Symbol
1 000 000 000 000 = 10 ¹²	tera	T
1 000 000 000 = 10 ⁹	giga	G
1 000 000 = 10 ⁶	mega	M
1 000 = 10 ³	kilo	k
100 = 10 ²	hecto*	h
10 = 10 ¹	deka*	da
0.1 = 10 ⁻¹	deci*	d
0.01 = 10 ⁻²	centi*	c
0.001 = 10 ⁻³	milli	m
0.000 001 = 10 ⁻⁶	micro	μ
0.000 000 001 = 10 ⁻⁹	nano	n
0.000 000 000 001 = 10 ⁻¹²	pico	p
0.000 000 000 000 001 = 10 ⁻¹⁵	femto	f
0.000 000 000 000 000 001 = 10 ⁻¹⁸	atto	a

* To be avoided where possible.

*MISSION
of
Rome Air Development Center*

RADC plans and conducts research, exploratory and advanced development programs in command, control, and communications (C³) activities, and in the C³ areas of information sciences and intelligence. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.

